Durham University

Durham E-Theses

A portability assistant for Fortran applications

Elizabeth Ann Gandy

How to cite:

Gandy, Elizabeth Ann (1993) A portability assistant for Fortran applications. Masters thesis, Durham University.

Use policy

The full-text may be used and/or reproduced, and given to third parties in any format or medium, without prior permission or charge, for personal research or study, educational, or not-for-profit purposes provided that:

- a full bibliographic reference is made to the original source
- a https://etheses.durham.ac.uk/id/eprint/5731/ is made to the metadata record in Durham E-Theses
- the full-text is not changed in any way

The full-text must not be sold in any format or medium without the formal permission of the copyright holders.

Please consult the full Durham E-Theses policy for further details.

Academic Support Office, The Palatine Centre, Durham University, Stockton Road, Durham, DH1 3LE
e-mail: e-theses.admin@durham.ac.uk Tel: +44 0191 334 6107
https://etheses.durham.ac.uk

A PORTABILITY ASSISTANT
FOR
FORTRAN APPLICATIONS

by

Elizabeth Ann Gandy
B.Sc. (Dunelm)

The copyright of this thesis rests with the author.
No quotation from it should be published without
his prior written consent and information derived
from it should be acknowledged.

Being a thesis submitted for the degree of
Master of Science
of
The University of Durham
1993

15 JUN 1994

ABSTRACT

This thesis addresses the issues of porting software from one machine environment to another. Some general observations are made about the definition of Portability and the design and portability of programs written in high level programming languages, in particular Fortran. Two areas of portability are considered in detail:

(i) Portability Criteria and Measures - The main criteria affecting the portability of Fortran applications are identified and possible measures of the effects of these criteria considered. A Portability Function is defined for obtaining a measure of the percentage portability of Fortran programs.

(ii) Portability Assistant - The use of existing analysis tools to obtain measures of the criteria affecting the portability of Fortran programs is considered. A portability assistant is provided in the form of an Ingres Relational Database, which holds the data obtained from these measures, enables the portability function to be applied to the application and assists in the porting of the application.

The methods of measuring the criteria affecting Fortran programs and the use of an Ingres database as a portability assistant is then applied to a particular example, the porting of NOMIS, a large manpower database.

ACKNOWLEDGEMENTS

I am particularly grateful to Mr. Malcolm Munro for supervising this project and helping me to formulate ideas into practical solutions and for his advice on the preparation of this thesis. I am also grateful to his wife for reading the script and correcting my spelling and punctuation.

The project would not have been possible without the co-operation of the NOMIS team and I would like to thank them, in particular Mr. M. Blakemore, for the permission to use their programs as examples.

I would like to thank my colleagues in the Computing Service for their moral support and their patience on the many occasions when I became a 'real user' and required their technical help. In particular I would like to thank Mr. J. Lindley for enabling me to combine this research with a full-time job, Mr. R. Gawley for his help with TeX, Mrs M.E. Hood for her help with Ingres and Dr. J.G. Roberts for his help with Uniras.

I appreciate the support given to me by all of my friends. Mark especially, has given me large amounts of technical support and helped me to keep things in perspective. The thesis layout is based on his TeX macros. I would also like to thank Margaret and Martin for encouraging me to start this project and Liz, Kesta and Susannah for their support throughout.

Finally I am grateful to my Parents, Catherine and Megan for their support and encouragement. I have written this thesis for them.

TABLE OF CONTENTS

Page

Acknowledgements i

Table of contents ii

Table of figures vi

Chapter 1. Introduction

1.1. Software Engineering 1
1.2. Definition of Portability and Terms Used 2
1.3. Why Write Portable Software? 3
1.4. Portability Criteria and Measures 4
1.5. Portability Assistant and its Application 5

Chapter 2. Portability

2.1. Introduction 6
2.2. Definitions and General Comments 7
2.3. Criteria Affecting Portability 14
2.4. Formulae 22
2.5. Recommendations and Language Standards 23
2.5.1. Language Standards 23
2.5.2. Recommendations 26
2.6. Examples 27
2.6.1. Porting Existing Programs 29
2.6.2. Designing for Portability 33
2.7. Summary 34

Chapter 3. Portability Criteria

3.1. Introduction 35
3.2. Discussion of Criteria Affecting Portability 35
3.2.1. Execution of 'DO' Loops 35
3.2.2. Free Format Source Statements 36

3.2.3. Format of Variable Names 36
3.2.4. Variable and Function Length Specifications 37
3.2.5. Data Initialisation 38
3.2.6. Order of Statements 38
3.2.7. Hollerith Strings 39
3.2.8. Alternative Subroutine Return Calls 39
3.2.9. Character Handling 40
3.2.10. System Specific Subroutine Calls 40
3.2.11. Internally Defined Filenames 42
3.2.12. Units Assigned at Runtime 43
3.2.13. Character Codes 43
3.2.14. Quote Symbols 44
3.2.15. Default Unit Numbers 44
3.2.16. Statement Labels Limit 44
3.2.17. Limit to the Number of Concurrent Units Open 45
3.2.18. Hexadecimal Data in DATA Statements 45
3.2.19. Z and Q Format Types 45
3.3. Summary 45

Chapter 4. Portability Measures

4.1. Introduction 47
4.2. Tool Requirements 47
4.3. Examples of Specific Tools 48
4.3.1. Compiler Warning Messages 48
4.3.2. FTNTIDY Program 50
4.3.3. EXTREFS Program 51
4.3.4. Awk 52
4.4. Portability Function 52
4.4.1. Portability Score 52
4.4.2. Portability Function 57
4.5. Criteria Counts and Weighting Factors 58
4.5.1. Execution of 'DO' Loops 58
4.5.2. Free Format Source Statements 59

4.5.3. Format of Variable Names 60
4.5.4. Variable and Function Length Specifications 60
4.5.5. Data Initialisation not in DATA Statements 61
4.5.6. Order of Statements 61
4.5.7. Hollerith Strings 62
4.5.8. Alternative Subroutine Calls 62
4.5.9. Character Handling Criteria 62
4.5.10. System Specific Subroutine Calls 63
4.5.11. Internally Defined Filenames 64
4.5.12. Character Codes 64
4.5.13. Double Quotes Symbol 65
4.5.14. Default Unit Numbers 65
4.5.15. Statement Labels and Unit Numbers Limits 66
4.5.16. Hexadecimal Data, Q and Z Format Codes 66
4.6. Summary 66

Chapter 5. Portability Assistant

5.1. Introduction 69
5.1.1. Relational Databases and Ingres 69
5.1.2. The Ingres Relational Database 71
5.1.3. Structured Query Language (SQL) 72
5.1.4. Ingres Applications 72
5.2. Structure of the Portability Database 73
5.2.1. Application Tables 73
5.2.2. Criteria Tables 75
5.2.3. Results Tables 77
5.3. Database Applications 77
5.3.1. Introduction 77
5.3.2. Applications 78
5.3.3. Observations 81
5.4. Database Programs 82
5.4.1. Introduction 82
5.4.2. Portability-Count Program 82
5.4.3. Portability-Order Program 84
5.5. Summary 85

Chapter 6. Application

6.1. Introduction 86
6.2. Overview of NOMIS 86
6.3. The Problem 87
6.4. Particular Portability Problems 88
6.4.1. Introduction 88
6.4.2. Data File I/O 88
6.4.3. ASCII/EBCDIC Character Codes 90
6.4.4. Character Variables in COMMON Blocks 91
6.4.5. Data Compaction 92
6.4.6. Overview 92
6.5. Portability Measures 93
6.5.1. Introduction 93
6.5.2. Criteria Counts and Scores 93
6.5.3. Hatton et al.'s Formula 93
6.5.4. Tanaka's Formula 94
6.5.5. Weighted Portability Function 95
6.5.6. Portability of Individual Subroutines 96
6.5.7. Overview 102
6.6. Summary 103

Chapter 7. Conclusions and Outlook

7.1. Introduction 105
7.2. Portability Measures and Criteria 105
7.3. Portability Assistant 109

Appendix 1. Sample Results from FTNTIDY 111

Appendix 2. Portability of NOMIS Subroutines 113

Appendix 3. Examples of Ingres Database Tables 122

References 126

Bibliography 128

TABLE OF FIGURES

		Page
Fig 2.6a.	Mobility of STYLE modules	30
Fig 2.6b.	Mobility of C-converted STYLE program	30
Fig 3.2a.	Non-portable length specifications	37
Fig 3.2b.	Order of Fortran Statements	38
Fig 4.3a	Example of an Awk script	52
Fig 4.6a.	Weighting Factors for Criteria	67
Fig 4.6b.	Weighting Factors for MTS Specific Subroutines	68
Fig 5.1a.	Possible database table structures	70
Fig 5.3a.	Mtsroutines Application Frame	79
Fig 5.3b.	Routines Application Frame	80
Fig 6.4a.	Sample results of method to detect ASCII/EBCDIC codes	91
Fig 6.5a.	Summary of counts of portability criteria affecting NOMIS	94
Fig 6.5b.	Frequency of portability values for NOMIS subroutines	97
Fig 6.5c.	Subroutines with the highest criteria scores	98
Fig 6.5d.	Subroutines with the highest number of lines of source code	99
Fig 6.5e.	Summary of criteria affecting subroutine R1	99
Fig 6.5f.	Summary of criteria affecting subroutine R2	100
Fig 6.5g.	Summary of criteria affecting subroutine R3	101
Fig 6.5h.	Summary of criteria affecting subroutine R4	102
Fig 6.5i.	Summary of criteria affecting subroutine R5	102

Chapter 1. Introduction

1.1. Software Engineering

Software Engineering is concerned with the design and development of software built by teams rather than individuals, this software is developed using engineering principles. The term includes both the technical and non-technical aspects of software development. For example project management and user problems are important aspects of Software Engineering. The term Software Engineering does not have one single definition. Sommerville [SOMM92] gives a number of common factors between the varying definitions.

The term software is considered as not simply the computer programs associated with the application, but also the documentation required to install, use, develop and maintain the programs. Well-engineered software has a number of important attributes. The four main software attributes are listed below:

(i) Maintainability - Software is subject to regular change so it should be written and documented so that the changes required can be implemented without undue costs.

(ii) Reliability - Software should perform as expected and should not fail more often than is allowed for in its specification.

(iii) Efficiency - Software should not make wasteful use of system resources such as memory and processor cycles. However, maximising efficiency should not be at the expense of making the software difficult to change.

(iv) Appropriate User Interface - The interface design should be taylored to the capabilities and background of the expected users so that it can be used to its full potential.

These attributes all have cost implications and optimising them is difficult. Some are exclusive, for example efficiency may be obtained at the expense of maintainability, so the optimisation is highly dependent on the particular requirements of the application considered. Maintainability is the key attribute since most software costs are incurred after the software has been put into use.

In order to measure the quality of well-engineered software there are a number of quality attributes that can be taken into account. Some of these quality attributes are [SOMM92]: Economy, Integrity, Documentation, Understandability, Flexibility, Interoperability, Modularity, Correctness, Reliability, Modifiability, Validity, Generality,

Testability, Reusability, Resilience, Usability, Clarity, Maintainability, Portability, and Efficiency.

It is the Portability attribute of software that is considered in detail in this thesis and a number of related attributes are:

(i) Understandability - Before a program can be ported it must be fully understood. Therefore, the portability of a program can be greatly affected by its level of understandability.

(ii) Modularity - The splitting up of a program into discrete modules can have an effect on portability. With a modular structure, those areas which are non-portable can be restricted to particular modules thus reducing the number of modules which will require modification.

(iii) Reliability - This may be considered the most important dynamic characteristic of a software system. It is a measure of how well it provides the services required of it and depends on how the software is used so cannot be specified absolutely. Reliability is important when considering portability and the resulting system should be at least as reliable as the original.

(iv) Modifiability - It is important when porting programs that the code has a high degree of modifiability. If the code is modifiable then the effort involved in porting it will be reduced.

(vi) Generality - If software has a high degree of generality the its portability will be improved since there is a greater chance that equivalent features will be available on target systems.

(vii) Reusability - This is the ability to reuse all or part of the software in another context. Portability is a particular form of reuse where the complete program is made available on another system.

1.2. Definition of Portability and Terms Used

It is necessary to determine exactly what is meant by the term 'Portability'. The literature survey in Chapter 2 looks at a number of alternative definitions of portability and other terms associated with it such as 'transportability' and 'mobility'. There is no general definition of portability or what is meant by portable software. The Oxford English Dictionary gives the following as a definition of the word portable:

"Portable: able to be carried or moved easily."

For the purpose of this thesis the following definition of portable is given:

> A program is said to be portable between two or more computer systems if the effort required to move the program and adapt it to work on the new system is less than the effort required to re-write it for the new system.

The terms 'portability' and 'mobility' can be considered equivalent and defined as follows:

> Portability (and Mobility) is a measure of the effort required to move a program from one computer system to another.

Throughout this thesis the term portability has also been used interchangeably with the term portable. It will be obvious from the context which meaning of portability is most appropriate.

For a measure of portability to be quantified it is necessary to restrict the systems under consideration to a finite set, since general portability (in the widest sense) is an unattainable target. There will always be a system (or one could be designed) that a particular program cannot be ported to. In this thesis, unless otherwise stated, general portability is taken to mean portability across a finite set of systems, namely all those systems having a Fortran 77 compiler available.

Another frequently used term is that of a 'target' system or machine. This denotes the system to which the program in question is being ported. A program can have one or many such target systems.

1.3. Why Write Portable Software?

The next point to consider is 'Why write portable software?' One answer to this question is given by Cowell [COWE77] in his introduction to papers from a workshop on the Portability of Universal Software organised by Argonne National Laboratory (1976). The following quotes are given as a starting point for the conference:

> " 'Those who cannot remember the past are condemned to repeat it.' - Santayana
>
> 'The man who doesn't write portable software is condemned to re-write it.' - J.F. Traub (1971) "

Some of the advantages of designing software with portability in mind are given in Chapter 2, together with some recommendations and advice on doing this. However

these recommendations are irrelevant in the situation where existing software has been written without portability in mind. It would be inappropriate to condemn these programs to be re-written completely. Most of them will have some degree of portability and it is the purpose of this thesis to provide an assistant for determining where portability problems (of Fortran programs) lie and in reducing the effort involved in porting them. By determining the portability problems in a piece of code it may also be possible to incorporate some of the recommendations given for designing portable programs and improve the future portability of the program.

1.4. Portability Criteria and Measures

Before obtaining measures of the portability of a program it is necessary to identify the areas where portability problems are most likely to occur. The most relevant of these criteria found in Fortran programs are described in detail in Chapter 3 and fit loosely into three areas of the Fortran programming language:

(i) Operating System Criteria.

(ii) Criteria concerned with the format of the source code.

(iii) Criteria caused by extensions to the Fortran Language Standard.

After the theoretical discussion of these criteria, Chapter 4 considers how the occurrence and effects of the criteria identified can be measured. This thesis does not aim to provide full static analysis tools for Fortran code so the use of existing tools and programs is considered. The main requirements for these tools are identified and some examples of appropriate tools given. These measures can be combined together to define a formula for the measurement of the portability of Fortran programs. This formula, a 'Weighted Portability Function' takes into account a weighted score given to the measure of each criterion affecting the code, the number of lines of source code and a constant, the 'Portability Factor', which is an indication of the levels of portability accepted for the program to be considered portable or non-portable. The results from this formula are given in the form of a percentage portability for the code under consideration.

Chapter 4 concludes with a detailed description of how the tools and programs previously identified can be used to detect the occurrence of each criterion affecting the code, together with the values which are considered most appropriate as weighting factors for obtaining the weighted score for the effect of that criterion on the code. These weighting

factors, ranging from 0 to 10, depend on the seriousness of the effects the criteria have on the code and the effort involved in detecting the occurrence of such criteria.

1.5. Portability Assistant and its Application

For large programs comprising a large number of subroutines, the measurement of criteria affecting the portability of the code can amount to a large amount of data. For this data to be used in assisting the porting of the program subroutines it is important to hold this data in a form which is easily accessible. Chapter 5 provides a solution to this problem, the use of an Ingres relational database to hold the information. It describes, in detail, a possible structure of such a database and how other Ingres utilities, for example 4GL Applications and embedded Structured Query Language (SQL) programs can be used to access the data and produce reports from it. One such embedded SQL program can be written to apply the weighted portability function to the information in the database and produce values for the portability of each component subroutine.

This approach of using an Ingres relational database as an assistant to portability is then demonstrated by its application to a particular example (Chapter 6). The particular example chosen is the porting of a large manpower database NOMIS (National Online Manpower Information System), written in Fortran, from an Amdahl 5860 running the MTS (Michigan Terminal System) operating system to a Sun Microsystems Sparc Server 2 running the Unix operating system. The structure of the NOMIS program, which was designed specifically for the MTS system, making full use of system specific facilities, is described in detail, together with the major problems encountered in porting the program. An Ingres database with the structure described in Chapter 5 was used as an assistant to the porting of NOMIS and allows detailed information about the criteria affecting all the component subroutines to be obtained. It also allows the weighted portability function to be applied to NOMIS so that a measure of its portability, and a comparison between the portability of its component subroutines can be obtained.

The thesis concludes with a summary of conclusions deduced from the previous chapters and a look at how this portability assistant could be developed and extended.

Chapter 2. Portability

2.1. Introduction

This chapter reviews some of the important points written about portability which are of relevance to this thesis. There does not appear to be one general definition of the term 'portability' so a variety of definitions are compared. In some cases the term portability is not used as such, with words such as 'mobility' and 'transportability' being used to mean much the same thing. Once portability (or its equivalent) has been defined it is then possible to break it down into a number of different types. These different types of portability are discussed together with some of the reasons for writing portable software.

Many of the papers give examples of specific problems found to be most common in porting programs and list the criteria they have found to affect portability. Of most relevance to this thesis are comments made about the portability of Fortran 77.

Due to the lack of a standard definition of portability there is no general formula for measuring the portability of a program. However, using particular definitions of portability some formulae have been defined which provide a measure of how portable a program is and some of these formula are described in relation to their use with particular example programs.

The definition of programming language standards is important when considering portability. Some of the difficulties apparent in defining a language standard are illustrated by a summary of an article on the development of a new language standard for Fortran. The adherence to a language standard goes a long way towards producing portable programs, however, language standards are not always fully defined (for good reason in many cases) so this review includes some of the recommendations and advice given on how to avoid or overcome these problems. It also considers what are the best ways of using system specific features, if they are unavoidable, to minimise the degradation in portability.

The chapter concludes with a number of examples of porting real programs. These show both how existing code (which has been designed for a particular system without considering portability) has been successfully ported and how efficient code can be implemented across a wide range of systems by designing it with portability in mind.

2.2. Definitions and General Comments

The first definition given is a definition of application portability given by Mooney [MOON90]:

> "An application is portable across a class of environments to the degree that the effort required to transport and adapt it to a new environment in the class is less than the effort of redevelopment."

Mooney defines two important aspects of portability as "transportation" which is the physical movement of the program and data to the new system and "adaption" which is the modification of the program and data necessary for it to work satisfactorily under the new environment. In this thesis we concentrate on the latter of these aspects of portability, the modification of the code (adaption) rather than the physical transportation of the program and data which was performed using straightforward file transfer protocols (FTP).

The emphasis given by Mooney is that portability is not a definitive quantity but more a "matter of degree". The degree of portability can be taken as loosely proportional to the class of target environments. If this class of target environments is restricted to a small set of similar environments then it would be expected that the level of portability would be higher than if the class of target environments were more general. It is true that over a general class of environments portability cannot be defined, however if this class of environments can be restricted to a finite set then the portability of the system can be defined and even quantified as this thesis shows.

Mooney states that portability can be divided into three types listed below:

(i) Binary Portability - the program can be transported to the target system in its executable form without needing to be re-compiled. This is the ultimate goal in any exercise in portability however it is very rarely the case.

(ii) Source Portability - programs which do not have binary portability can still be considered to have a high degree of source portability if the source code can be transported to the target system and re-compiled using a compiler more specific to that system, with the minimum of changes necessary to the source code.

(iii) Experience Portability - this can also be called "user portability" or "personal portability" and is the experience gained by porting similar programs and systems, thus reducing the human effort required in porting the program in question. A program

which does not exhibit binary or source portability may be considered to have some degree of experience portability if it uses standard user interfaces, file structures and software tools. Experience portability may be increased by using portable tools and portable systems software to implement user interfaces and file systems.

Lemoine and Muller [LEMO81] give definitions of portability and transferability, another term which occurs frequently in discussions on portability. In many cases the differences between these two terms are not defined. Lemoine and Muller define:

"*Portability* characterises the choices made at the moment the software is *designed* so as to carry out later completely automatic transfer operations.

Transferability defines the degree of software portability according to the importance of the choices dependent on a given machine"

From this it can be seen that there is a potential conflict between these two terms since what in this case is defined as transferability is a very similar definition of what Mooney considers to be portability. This definition could be taken as rather restrictive in considering the portability of a program to be set just at the design stage since it is continually possible to improve the portability of a program by re-designing particular features.

Hague and Ford [HAGU76] consider what was in its day (1976) a new approach to portability. Two approaches to the portability problem are defined, the first which has traditionally been the approach used is "corrective portability" where software would be written for one machine without taking any other machine into account. If the software was subsequently to be ported to another machine then the source code would be modified by "trial and error" until it appeared to be working. The second approach, which they consider in detail in their paper, is that of "predictive portability" where previous experience is used to restrict the programming language used to a subset which is, as far as possible, available on both the original machine and any machines to which the program is likely to be ported. This approach will obviously only work in the situation where it is known at design stage which machines the program needs to be ported to or that the language subset is governed by a standard whereby it is generally available on a wide range of machines.

Using previous experience it will be known what changes will need to be made to the code to transfer it to subsequent machines and these changes, or variants, can be placed

in a composite version of the program. A pre-processor can then be written to extract the appropriate set of records for the particular machine required. This approach is most useful in the situation where the code is being developed on one machine, then exported to other machines as production versions. As subsequent versions of the software are developed they will need to be ported to all the machines required, so the workload will be reduced drastically if this process can be automated.

If the necessary changes to the code could have been predicted then Hague and Ford state that:

> "...In this case we could systematically *trans*form the original code and *port* the modified software directly to the target machine. This, in our view, would constitute transportable software"

then go on to formalise this definition:

> "A program is *transportable* between a specified set of machines if the changes necessary to satisfy specified performance criteria are capable of mechanical implementation via a software processor."

Hague and Ford also consider some important points about portability and transportability. The most important constraint on the portability of software is considered to be the range of systems for which it is to be made available. It is indicated correctly, that "Universal Portability is an unattainable target" and that no matter how definitive a software standard, a system can always be invented (even if it doesn't already exist) that will violate this standard. In any case where actual values are given to portability it is necessary to restrict the systems considered to a finite domain, even if this domain is large. If no changes are necessary when moving a program to any machine within this domain then the program can be considered "completely portable" within that domain. When considering the criteria affecting the portability of Fortran later in this thesis a domain of machines (other than the set of those that have Fortran 77 available to them) has not been defined specifically, however there are many criteria that we consider which violate general portability (i.e. there is at least one known machine where the particular construct does not work or works in a different way) but if we were to restrict the domain of machines to, in some cases, a very small set then these criteria are not detrimental to portability.

Tanenbaum et al., [TANE78] give a more quantitative definition of portability in terms of human effort, defining portability as:

"...a measure of the ease with which a program can be transferred from one environment to another; if the effort required to move the program is much less than that required to implement it initially, *and* the effort is small in absolute sense, then that program is highly portable"

This is a very broad definition of portability and does enable a measure of portability to be defined. However, it does not take into account the variations between different environments and so will always be specific to a particular set of environments being considered. Also, since this definition takes into account some measure of the effort required to move the program, the portability of the program cannot be quantified before moving the program and so cannot be used in planning resources for porting the program. However, this is a good definition of portability for the case where software is being continuously developed on one machine and production versions ported to subsequent machines. Once the initial port has been done, a measure of portability can be given, which may be different for each target machine, and this measure can be helpful in porting subsequent versions. This could be considered more specifically a definition of experience portability described by Mooney.

Hatton et al., [HATT88] discuss the design of portable software and begin by giving a number of reasons as to why software should be designed with portability in mind:

(i) Portability allows the early use of new technology since it is more likely that the use of portable languages etc., will be quickly available on any new systems developed. This also helps in price-performance since a portable program is more easily moved to a different system if it is considered more economical in both costs and performance to move to that system. If the software is not portable then the cost of porting it may outweigh the savings gained from the new system.

(ii) Portability allows the software to be maintained across different machine types during hardware transition periods without too much of a burden on software maintenance personnel.

(iii) Portability increases the commercial possibilities of marketing the software on many different systems without an excessive load on the maintenance of the software.

Hatton et al. discuss the design of a large Seismic Kernel system which was designed very much with portability in mind. Few (if any) programming languages are completely portable in practice, however at the start of this project (1982) Fortran 77 was considered

the best choice because of its good portability, efficiency and availability. Hatton et al., feel that at the time of writing (1988) the decision would still be the same, C is now as widely available as Fortran but its main disadvantage was the tendency to produce write-only (human unreadable) code. Also, its default floating point computation is double precision which would reduce efficiency in their particular project as that level of precision was not required. Pascal was also rejected since it has a very restricted language definition and so is essentially non-portable.

Hatton et al., found, through their project that portability could be split into a number of different types:

(i) Intrinsic Portability - the portability of the actual programming language itself which they found was related to the simplicity of the language and the quality of its standard definition.

(ii) Conceptual Portability - the case where the code itself may comply with the language definition and so will be intrinsically portable, however, the use to which it is put may be highly system dependent.

(iii) Peripheral Portability - the portability of peripheral devices used such as accessing the disk filing system.

Another consideration in designing portable software, identified by Hatton et al., is the choice of a user interface and how portable that will be. They consider two options:

(i) Menus - These are an attractive option in many cases but are difficult to design efficiently without resorting to a full WIMP (Windows, Icons, Mice, Pull-down menus) system.

(ii) Problem-Oriented Language - With this kind of user interface a language is designed for the application concerned and programming requests are coded in this language. The main program must then include a command interpreter for this language.

Provided the application environment and underlying model are simple enough for the command language to remain simple then the latter option would be the most portable. User interfaces are not considered in detail in this thesis but it may be interesting to note that the NOMIS example described in Chapter 6 makes use of a problem-oriented language as its user interface.

It can be argued that portability and efficiency cannot be achieved simultaneously since for a program to be efficient machine dependence should be exploited. Hatton et

al., reject this argument to state that "portability and efficiency not only do, but *must*, go hand in hand in large systems design." Obviously there are situations where it may be necessary to use non-portable constructs to attain maximum efficiency for the code and alternatively there are situations where it may be more beneficial to choose a less efficient construct where there is a choice between one that is portable and one that may be efficient on one system but unavailable on another. The important point is to strike a balance between the two. Hatton et al., formalise this by defining a hypothesis which is given below but not discussed further since we are considering portability rather than the arguments for and against portability and efficiency.

> "In large system design, if locality is the major architectural consideration, portability (locality of device dependence), efficiency (locality of activity dependence) and reliability (locality of intellectual dependence) should quite reasonably occur simultaneously, even if the locations are different."

In this case intellectual dependence is the organisation of program development so that the programmer considers local rather than global issues at any one stage.

Reinsch [REIN77] considers some of the positive side effects which can be achieved in striving for portable software. These can be summarised as follows:

(i) It revives the interest in the standardisation of programming languages.

(ii) If programmers are not willing to use system dependent extensions to programming language standards such as Fortran 77 because of their concern for portability then it is more likely that they will press for extensions to be made to the standard, thus improving functionality and efficiency.

(iii) It creates an increased awareness of a potentially varying environment so that if non-portable constructs are used, they are used with care and well documented.

(iv) It provides a greater acceptance by programmers of rigorous and stringent test routines, including performance evaluations of different types of machines.

(v) It makes programmers reluctant to correct flaws in machines by writing system specific software. If programmers refused to (or were prevented from) using software measures to avoid flaws and pitfalls in say, machine arithmetic then they would be compelled to use 'clean computers'. By Darwin's law of evolution, if this was the case, only these 'clean computers' and their manufacturers would survive, thus improving the quality of machines available.

(vi) It gives programmers a greater awareness of machine parameters and arithmetic prerequisites.

Smith [SMIT77] gives alternative definitions of both portability and transportability:

"*portable program;* a program that can be compiled and will execute correctly without change at each of the computer installations under consideration.

transportable program; a program that is sufficiently structured that it can be ported from place to place by making only automated changes."

With this definition portability can be taken as a subset of transportability.

Wallis [WALL82] considers the design of programs with portability in mind, giving the following definition of what he considers to be a good portable program:

"A good portable program design is one that suits any portability method, even the complete recoding of the program from documentation produced in the course of an earlier implementation for another machine."

This is a much wider approach to portability. Previous definitions of portability have, one way or another, been concerned with moving programs from one machine to another with the minimum of change. Wallis considers that, while this is the ideal, there are times when it is necessary to rely on machine dependencies but provided the program is well-designed this does not render the program completely non-portable. A number of reasons when it may possibly be necessary to use machine dependencies can be given:

(i) Programs handling character information which may require some form of system dependent packing.

(ii) To utilise the full range and accuracy that a particular machine can provide.

(iii) The need for system specific features of the run-time environment such as special operating system routines or the use of overlaying techniques.

(iv) The use of machine dependent I/O facilities for files having special formats.

(v) The need for special features to improve run-time efficiency.

It may seem surprising that in a book concerned with the design of portable software, the use of system specific facilities is advocated but Wallis is considering the practical aspects where he feels that occasionally designing for portability can be a disadvantage. It is stated that that the programmer "can write programs in a conservative way to avoid

problems or trade some portability in exchange for writing a less severely constrained program". It is considered a "question of priorities".

2.3. Criteria Affecting Portability

Mooney [MOON90] looks in detail at the use of standard languages, translators and libraries and identifies a number of high level operations which are typically omitted from programming language standard definitions since they are highly system dependent. Examples are given of such operations as string handling, numeric algorithms and file and I/O processing. In many cases these system specific operations are provided as procedures supplied through standard or portable libraries. Provided the same libraries are available on the target machine, the use of such procedures make the programs more portable than if they had been written using system specific routines. In the general case the use of libraries will make the program non-portable, however in practice, provided the set of target machines is such that they all have these libraries available, there will be no detriment to portability. One example of such a standard portable library is the Numerical Algorithms Group (NAG) library for use with Fortran programs which is widely available.

Tanenbaum et al. [TANE78], consider the general areas in which programs are not likely to be portable. These areas are summarised in the following list although it is emphasised that some of the areas overlap:

(i) Programming Languages.

(ii) Real Numbers.

(iii) Files.

(iv) Physical Media.

(v) Interactive Terminal I/O.

(vi) Operating System.

(vii) Machine Architecture.

(viii) Documentation.

Each of these areas are described in detail by Tanenbaum et al., but it suffices here to consider the problems identified under the heading of programming languages:

(i) Dialects - Not all systems accept exactly the language standard, assuming one exists. In some cases only a subset of the language may be implemented thus meaning some code may not compile even if it complies exactly with the language standard.

(ii) Identifiers - The set of allowable layout symbols such as space, newline and tab may differ between implementations. Some languages allow identifiers (variable names) to be an arbitrary length, however most compilers have some form of restriction and these restrictions differ between compilers. Some accept an arbitrary length but only discriminate on the first N characters (a common value for N is 6).

(iii) Stropping - The begin symbol in some languages (such as ALGOL 68) is printed in bold in publications, however its computer representation is not standardised, common representations being "begin", "begin, BEGIN etc. Some implementations use reserved words instead of stropping. Therefore, the use of identifiers which are the same as these possible reserved words like begin should be avoided so that the program text can be transformed to a representation without stropping with a macro-processor or text editor if required.

(iv) Pragmats - Some languages (eg. ALGOL 68) allow certain commands, hints and advice to be provided to the compiler. These are known as pragmats and their syntax and semantics are highly compiler-dependent.

(v) Mapping of types onto machine words - A compiler with byte addressing and a 16-bit word may use 16-bit integers or 32-bit integers (or some other length). Thus different compilers for the same language may have different length integers, reals etc. This can be a problem with Fortran COMMON variables since changing the size of variables make previously correct COMMON declarations semantically incorrect.

(vi) Separate Compilation - Some compilers allow procedures to be compiled independently which may be useful when compiling very large programs. If such a program were moved to a system where the compiler did not allow separate compilation then the program may not compile because it is too large. Also separate compilation is not standardised in most languages so there may be differences in the order in which separately compiled procedures are tied together with the linkage editor.

(vii) Maximum array and string sizes - Some compilers have limits on the maximum number of elements in an array, the maximum subscript value, the maximum number of subscripts or the maximum length of string constants.

(viii) Packing - There may be problems porting programs which make assumptions about how many characters fit into an integer. Also some compilers pack the elements of Boolean arrays into words with one element per bit whereas others pack one element into a whole word.

(ix) Run-time checking - Most languages require some run-time checking such as array bounds, initialised variables etc. but in most compilers these checks can be turned off to improve execution speed. Apparently there have been cases where programs ran 'perfectly' when this checking was turned off but gave errors when the checking was enabled.

Under the heading of programming languages, Tanaka [TANA92] identifies the following criteria which have been found to reduce the portability of Fortran between different systems:

(i) Maximum array and string sizes.

(ii) Maximum length of identifier.

(iii) Range and Precision of reals.

(iv) Format of 'Include' statements.

(v) Format of 'Block Data' statements.

(vi) Recursive calling methods.

Tanaka notes that problems are more apparent in Fortran than in C due the incomplete language specification of Fortran in the past and concludes his paper with the statement that "many problems arise due to the variants of Fortran compiler, and most of them can be eliminated by converting those modules to C language." It could be argued however, that while these problems may be eliminated by converting the modules to C, other problems may become apparent later due to the fact that C also has a loose language definition.

Hatton et al., [HATT88] give a particular example of a portability problem which affects what they define as "peripheral portability." In the case of disk filing systems the Fortran 77 standard is unsatisfactory and a statement such as:

```
OPEN (UNIT = UNNUMB, FILE = FLNAME, ...)
```

complies with the standard but both UNNUMB and FLNAME are completely implementation dependent. FLNAME contains all the information about the file's identity with

regard to the local filing system. This was a particular problem with the NOMIS program described in Chapter 6 as filenames on the MTS system were of a different format to those on the target Unix system. Unit numbers are also a problem since the only restriction imposed by the Fortran 77 standard is that they are non-negative, whereas different implementations reserve different unit numbers for frequently used devices.

Hatton et al., make a number of further observations about "practical portability" found during their porting of the SKS Seismic Kernel System. It was found that approximately two thirds of the Fortran 77 standard had no portability problems, however, the following statements caused problems at some point while porting the SKS system:

BACKSPACE, CHARACTER, CLOSE, DIMENSION, ELSE IF, ENDFILE, ENTRY, INQUIRE, OPEN, PRINT, READ, REWIND, WRITE.

The most comprehensive discussion of portability criteria is given by Larmouth [LARM81]. In addition, this discussion is most relevant to this thesis since all the criteria are problems with Fortran 77. It could be said that any program which is not deviant from the Fortran 77 standard should run and produce identical results on any system conforming to the standard. This may not be the case since there are a number of areas which the standard does not attempt to define, for example, the maximum length of arrays, the depth of nesting loops and the size and number of program units.

These areas which are not defined by the standard should be considered as separate from those areas of deviation from the standard. Common deviations are the use of manufacturers language extensions, division by zero, access beyond the end of an array and the use of 'undefined' values.

The main areas which the standard does not define are summarised below:

(i) The problem of processors treating an external as a local intrinsic. This can be resolved by the use of EXTERNAL.

(ii) The problem of hardware precision. The advice here is to use system-provided subroutines even though this will be non-portable.

(iii) The area of INPUT/OUTPUT contains a lot that is not standardised and also some areas where it is not clear what is actually included in the standard.

(iv) The model of the file store is not standardised.

(v) The collating sequence used for character comparisons is only partially standardised.

Larmouth discusses some specific problem areas in more detail. The main points are included below:

Real Values: Results involving reals should not be too dependent on a certain degree of precision or method of rounding since these may differ between different implementations of Fortran. Flow of program control should not be unduly dependent on these factors (for example using IF (A.EQ.B) GOTO ... where A and B are reals) and the use of real variables in DO loops should be avoided. Also, the internal representation of a constant is not standardised.

Storage Units and Boundaries: The standard has two sorts of storage units, one for numerical values and one for character values. These should remain wholly separate if the standard is adhered to and character and numerical values should not be EQUIVALENCED or should not be in the same COMMON BLOCK. The type specifiers DOUBLE COMPLEX and INTEGER*2 are not included in the standard, however all (according to Larmouth) implementations allow them. The use of these can have serious portability problems as they are interpreted differently by different compilers. Some compilers cannot handle arrays which cross a 'page' or 'segment' boundary. In practice this means that for portable programs, common blocks and arrays should be kept under 8000 numeric (or 32000) character) storage units long.

Character Handling: Lower case letters are not included in the fortran standard and whilst it is not deviant from the standard to use any characters in a comment it is not recommended that the Fortran character set be exceeded even in comments if true portability is required. The results of the functions CHAR and ICHAR are not standardised but with care can be used in portable programs for decoding output. It should be safe (but not guaranteed by the standard) to use them for the Fortran character set but their use is not recommended for the full ASCII character set. In the NOMIS example, after serious problems converting from EBCDIC to ASCII, the use of CHAR and ICHAR was recommended over the use of the actual numeric character codes.

Although some implementations allow zero length character constants they do not conform to the standard. Also Fortran strings are always fixed length so there is no distinction between:

```
CHARACTER*4 A
A = '1'
```

and

```
A = '1   '
```

The use of the LEN function will always give 4. Thus the use of zero length strings to represent nulls is not possible. There has apparently been recorded a case of one compiler which allowed strings to be "variable length subject to a declared maximum" as a language extension but this is uncommon and should be avoided in portable programs.

Use of Functions:

Three main points are noted in this area:

(i) A function call cannot affect the value of any variables or the value which will be returned by any other function used anywhere in the statement containing the function call. This restriction enables compilers to evaluate functions in an expression in any order.

(ii) The arguments of a statement function cannot be changed either directly or indirectly.

(iii) Effects which depend on a function being called when its value is not actually needed produce 'undefined' values. This is likely to make the program deviant later.

INPUT/OUTPUT: Another area in which portability problems occur is in the errors produced from I/O operations. One such error is that there is no standardisation on when an I/O error condition ceases to exist. If an error occurs in an I/O operation with ERR= and/or IOSTAT= present then there is a discrepancy over when a subsequent INQUIRE statement (which wouldn't in itself be erroneous) will report an error. Some such differences between compilers are listed below:

(i) INQUIRE will never report an error since an error ceases to exist as soon as the statement producing it is completed.

(ii) INQUIRE will report the error once if ERR= was used in the original I/O statement but not if IOSTAT= was used as well since the error ceases to exist as soon as an IOSTAT is set.

(iii) INQUIRE will report the error continuously until a subsequent I/O operation is performed on that unit.

End of file conditions also cause portability problems particularly when no endfile record exists. The advice is to use END= to trap the reading of records which don't exist even though this is deviant from the standard.

Larmouth concludes with a list of common extensions which, although available with many compilers, are not generally portable:

(i) INTEGER*n.

(ii) Character and integer variables mixed in COMMON blocks or EQUIVALENT statements.

(iii) Multiple IMPLICIT statements in a subprogram.

(iv) Multiple confirmatory 'typing' of a name.

(v) Multiple definitions of a PARAMETER (replacing a previous definition).

(vi) Relaxing some or most of the restrictions on the ordering of statements.

(vii) COMMON blocks and PARAMETER names the same.

(viii) Alternative returns in functions.

(ix) Intrinsic functions used in constant expressions.

(x) Recursion.

(xi) Provision of extra local keywords on OPEN.

(xii) Overlapped character assignments, overlap of FORMAT and input variables on READ.

(xiii) 'Extended range' of DO loops.

(xiv) Hollerith constants.

(xv) Unsubscripted array name used as an actual argument when the dummy argument is a simple variable.

(xvi) RETURN (meaning STOP) in a main program.

(xvii) General expressions in statement functions, in particular substring references.

Smith [SMIT77] considers the criteria affecting portability to be a "poison" defining a "Fortran Poison" as an "unclear non-transportable Fortran construct" and giving three sources of such poisons:

"(i) Poorly structured representations of computational processes and tricky coding.

(ii) Inconsistent formatting conventions.

(iii) Constructs in Fortran that are either ambiguously defined or not defined by the ANSI Standard Fortran."

Wallis [WALL82] compares portability criteria affecting a number of different high-level programming languages. The conclusions made are best summarised by the following quote:

> "In broad outline the problems of portable programming in high-level languages seem almost independent of the particular language used; the problems usually include character set, language subset identification, permissive language standards, input/output problems and concerns with arithmetic range and accuracy."

While not completely relevant to this thesis Seacord [SEAC90] raises an interesting point about portability. Higher level languages such as C and portable operating systems such as Unix have increased the ease with which portable software can be written and it is becoming more common for software to be ported to different architectures with the only requirement being the need to recompile the software for that system. Seacord notes however that:

> "Porting an application to simply run on a different platform is not sufficient; the application interface is required to behave much like it does with other applications developed for that environment."

Seacord uses as examples of this the fact that if an application were to be ported to an Apple Macintosh then it would be expected that the user interface would be able to perform a variety of operations using a single-button mouse. If the same application were ported to an MS-DOS machine then it would be expected that the user would be able to perform the same operations using the keyboard.

If software is written to provide these specific user interfaces then this is a serious criterion affecting its portability. It is now necessary, when porting software, not just to consider whether it will compile and run on the new system but also how the software can be integrated into the user-interface toolkit of the new system. Seacord goes on in his paper to consider this portability criteria in more detail, suggesting that the application is separated from the user interface as much as possible so that changes can be made to the

user interface without modifying the application software more than is necessary. The use of User Interface Management Systems which perform this task is considered.

2.4. Formulae

Although it has previously been stated that an exact measure of portability cannot be attained, some papers do attempt to define a formula for portability. In these cases the formulae have been defined using the experience gained from particular examples and where the set of target machines is restricted to a finite set of specific machines. In this case it is possible to quantify portability to some extent.

From their experience with the Seismic Kernel System (SKS) Hatton et al., [HATT88] define the following formula:

$$mobility = 100 \times [1 - \frac{(time\ taken\ to\ port)}{(time\ taken\ to\ write)}]$$

with the following taken as reasonable measures:

portability is equivalent to $> 95\%$ mobile

transportability is equivalent to $> 80\%$ mobile

effective non-portability is equivalent to $< 60\%$ mobile

Tanaka [TANA92] also considers the use of this formula, however, the STYLE program considered in his paper was developed over several years making the 'time taken to write factor' difficult to define. It was found to be more appropriate to base the formula on 'lines of source code' and so the following formula can be defined:

$$mobility = 100 \times [1 - \frac{(number\ of\ modified\ LOC)}{(number\ of\ total\ LOC)} \times \alpha]$$

where LOC is 'lines of code' and α is a factor to normalise the difference in workload between developing and modifying the code defined as:

$$\alpha = \frac{workload\ for\ modifying\ 1\ LOC}{workload\ for\ developing\ 1\ LOC}$$

and assumes $\alpha = 3$ in the case of the STYLE program. Tanaka takes the acceptable measures of mobility for a program to be portable, transportable or essentially non-portable defined by Hatton et al.

These formulae are suitable for obtaining a rough estimate of mobility, the first only being really useful in the case where a detailed history of the program development is available. If a program has been continuously developed over a number of years as in the case of the STYLE program it may not be possible to define accurately the time taken to write the program. Tanaka's formula will give a much more accurate indication of the program's mobility. The drawback with this formula is that Tanaka assumes that the number of modified lines of code is a sufficient indication of the workload involved when in practice the workload may differ significantly between modifying one line of code containing a simple problem and modifying one line of code containing a complex problem. It will be seen later that Tanaka's formula can be modified further to allow the number of modified lines of code to be replaced by a weighted score proportional to the number of lines of modified code and a weighting factor for the particular criterion affecting those lines of code.

2.5. Recommendations and Language Standards

2.5.1. Language Standards

There has been a lot of discussion on the development of standard definitions for programming languages. The problems associated with defining a language standard for Fortran are discussed in an article in Computer Weekly (3/5/90) [COMP90]. The problems highlighted by this article are summarised below.

A Fortran standard, Fortran 77, already exists but this was agreed on mainly to standardise something rather than nothing. In 1982 Fortran 82 was set up as an intended definitive standard. Its starting point was to be a 'core plus modules architecture'. It was intended to have a core as a complete purpose Fortran and an 'obsolete features module' to contain obsolete and redundant Fortran 77 features so that existing Fortran 77 programs would still work. The weakness with this idea was in achieving agreement on which features should be in the core and which in the 'obsolete features module'.

This 'core plus modules' architecture was dropped in favour of Fortran 8X. In 1984 IBM and Digital Equipment voted against the publication of a draft progress report by the US ISO Fortran standardisation Committee, however, a draft standard was produced and went to an internal ballot within the ISO and ANSI Groups. The vote was fairly favourable except in the US (claims were made that the language was too large).

The US committee produced a 'compromise plan' with cuts in the draft standard. Large sections were deleted or put into suggested extensions. The reaction to this from outside the US was one of outrage.

Much work was done during 1986-7 and successive versions of the next draft were produced. By August 1987 there was almost a draft International Standard. The traditionalists were opposed to this plan and claimed that it was not compatible with Fortran 77. They incorrectly accused the free form source code (coding layout restrictions imposed by punch card technology could now be lifted) of invalidating millions of lines of existing code. During the late 1980's a lobby was set up within the US group.

When the first public draft was reviewed there were three types of negative reaction:

(i) Omission of 'pet' features.

(ii) Revisionists bemoaned missing the opportunity to fix a particular deficiency.

(iii) Technical objections.

There were also complaints that the language was too large.

The in-fighting continued with a vote on five major issues. All these won a majority but the package as a whole was voted down.

In Paris in the autumn of 1988 the International Group WG5 launched a framework for the new 8X standard. This was likely to achieve the support of all the member countries (except perhaps the US). It set a timetable for a second public review and declared Fortran 8X to be 'Fortran 88'. Early in 1989 IBM dropped its objections to Fortran 8X and the way looked clear.

In the summer of 1989 a vendor representative on the US X3J3 Fortran went over the heads of the committee and proposed Fortran 77 be retained alongside Fortran 8X instead of replaced by it. This was approved in the autumn. The traditionalist/vendor faction had convinced the committee that the users did not want Fortran 8X.

The suspected real reason for this lobby was that US IT procurements are normally based on conformance to standards. Vendors would contrive to get Federal contracts to specify Fortran but they would be left to supply Fortran 77 or Fortran 8X as they wished. The conformance rules of Fortran 77 are weak so they could supply Fortran 77 implementations with vendor-specific extensions incompatible with Fortran 8X. Although Fortran 77 is a subset of Fortran 8X, vendor implementations are generally not.

This is purely a 'domestic' problem since within the ISO any member country can specify what it likes for purely domestic standards. However it was argued that the US has a very strong influence on the rest of the Computer Industry.

In London in March 1990 the ISO WG5 worked on the latest draft of the now Fortran 90 standard and sent it back to the US X3J3 to finish. Current indications are that X3J3 will agree a final Fortran 90 draft which will be the same inside and outside the US. (The Fortran 77 issue is out of ISO control).

The committees referred to by this article are:

US Fortran Standards Committee X3J3 - technical work of producing the standard (US National and International) - Reports to X3, a committee of ANSI, the US member body of ISO.

International Fortran Standards Working Group WG5 - coordinates international comment on work of X3J3 and gives general advice on the direction the development of standards should be heading. The US members are drawn from X3J3.

Hunter [HUNT90] discusses the inadequacies of Fortran 8X claiming that "The X3J3 Committee is trying to reconcile two confliction purposes" and defines these purposes as the need for compatibility with Fortran 77 so that programs written in Fortran 77 will still compile with the Fortran 8X compiler and the need to add new features to enhance the language, correcting the original design flaws and adding new features that will bring Fortran up to date with other languages.

Hunter suggests that these are the main reasons for the complexity of Fortran 8X but are of vital importance when considering portability. When new versions of a programming language are defined there can be serious portability problems if upwards compatibility is not kept. If this was not the case then there would be serious portability problems, not only when moving between systems but also between different versions of the same language. When Fortran 77 was defined it remained compatible with Fortran 66 although this does increase the complexity of the new version and introduces conflicts and redundancy. The existence of Hollerith strings in Fortran 77 which are redundant due to the increased availability of string manipulation is one example. In a similar way there are eight different types of array declaration in Fortran 8X and yet only two are essential, the others included for compatibility with Fortran 77.

2.5.2. Recommendations

Many of the papers considered give advice and recommendations for both designing software for portability and improving the portability of existing software. This section includes details of these recommendations.

Mooney [MOON90] describes some possible strategies for designing and implementing portable software. In summary these strategies are:

(i) Strategies that maintain identical execution time interfaces by porting system components that form the interface (the use of portable translators, portable libraries and portable operating systems).

(ii) Strategies that maintain identical, or nearly identical, interfaces for different system components by adhering to appropriate standards for languages, libraries and operating systems.

(iii) Strategies that assist in the adaption of programs to a target environment (automatic translation, dynamic adaption and "designing for portability").

These strategies are broad but the most important could be considered to be that of designing software with portability in mind. This would, by definition, lead to the use of the other strategies and the use of standard and portable tools which may be available.

Mooney discusses the use of standard languages, translators and libraries indicating that historically, the use of common programming languages has always been the first step in designing for portability since it is more likely that translators for a common programming language may be available on many target systems. It is noted that the development of Fortran in the 1950's made this type of portability possible for the first time and that standardisation of programming languages, both formally and those established by common programming practices, have increased this portability.

Mooney concludes with a number of points which should be considered when designing software with portability in mind:

(i) A standard programming language should be used and non-standard extensions of the language should be avoided.

(ii) Standard or portable procedures should be used wherever possible for additional required functions.

(iii) A modular programming structure should be used so that any sections of the code which are non-portable can be kept in separate modules making them easier to re-write without affecting too much of the main body of the code if the program is to be ported.

Wallis [WALL82], whilst advocating the occasional use of system specific features warns against "accidental machine dependencies" where a programmer may make assumptions about the way in which a machine may react to a certain situation. These dependencies are particularly hard to detect since they will not appear until the program is running on a different machine. The use of a modular program structure is recommended in all cases but particularly in the case where system specific features have been used. If system dependent parts of the program are restricted to a few modules of the whole program then their replacement on future target machines may not cause serious problems. It may require more effort in the design stage to produce modular programs but ultimately they are better structured and easier to maintain.

Tanaka [TANA92] states that to enhance the portability or transportability of software, attention should be given to programming languages, operating systems, file systems, inter-process communication mechanisms, I/O device characteristics and machine architecture. An improvement in transferring methods for distributed software is also described involving the use of a computer network, having one centralised master source file which is a composite file containing the source code for each machine on the network and all the information for porting the code to the other machines on the network. A pre-processor on each machine on the network can then be used to extract the appropriate code for that machine.

Lemoine and Mullor [LEMO81] recommend that when designing software with portability in mind, if possible a high level programming language should be used. However they state that if such a language is inadequate in dealing with the particular problem then it may be possible to define a specific language, write the software in this language and then use general tools such as LDSS (English translation - Syntax and Semantic Descriptive Language) or macroprocessors to perform the translation to a real machine.

They conclude that:

"the fundamental problem to be solved is the evaluation of software dependence with regard to the original environment (machine, operating system, language):

This evaluation allows the definition of the degree of portability (or transferability) of the product."

Larmouth [LARM81] discusses Fortran 77 portability and recommends that in cases where the situation to be considered is not standardised by the language standard it may be necessary to use deviant code. This code should be segregated from the main body of the code and well documented to indicate that it may affect the portability of the code. Having considered in detail the problems of Fortran OPEN and INQUIRE statements (described in Section 2.3) it is further recommended that portable programs should make as little use of OPEN statements as possible and avoid INQUIRE statements. All OPEN and CLOSE activity should be segregated into a separate sub-program and well documented. The treatment of elaborate I/O facilities of Fortran 77 as a non-standard but widely available language extension is recommended and they should be segregated and their use documented. 32 guidelines are given at the end of the paper for portable programming in Fortran 77. The list is too long to be reproduced here but is invaluable to anyone planning to design portable programs in Fortran 77. Despite the problems noted in his article Larmouth concludes:

"Fortran remains one of the most fully defined languages we have available, and is certainly the only one with a high degree of portability"

This was written in 1981.

In the conclusion to their article Hatton et al., [HATT88] make what could be considered the most important recommendation to those designing portable software:

"The temptation to over-complicate is one of the biggest traps facing scientific program designers. SKS succeeded because its simplicity was enforced ruthlessly as much by the continuing emphasis on portability as by any conscious effort on the part of the designers. When the designer must design across several machines, only simplicity ever works."

2.6. Examples

Many of the points discussed in the previous sections of this chapter have been deduced from personal experience by the authors of porting software between a variety of systems. Some of the examples they have used are given below and can be split into two categories: porting existing programs which may have been written for a particular

machine and designing programs specifically for portability. Examples of both of these categories are given below:

2.6.1. Porting Existing Programs

Tanaka [TANA92] considers the portability of a fourth generation language system called STYLE, designed to speed up the application program development for business transaction processing. The main content of the article is an evaluation of the differences in portability of Fortran and C and how the portability of STYLE was affected by converting the Fortran modules to C.

The main problem encountered in the porting of STYLE was the extensive use of recursive calling methods which are available in some Fortran compilers but not others. The eventual solution to this problem was to re-write the offending routines in C. Other notable problems found with the porting of STYLE are given in the article, the most relevant (since they are similar to problems occurring in the NOMIS problem described in Chapter 6) being that STYLE uses a synchronous file I/O facility which has many variants depending on the operating system being used. This problem can be compared with that of direct access of variable length records used in the NOMIS program on the MTS system. Another problem was that of user authentication when porting to single user operating systems such as OS/2. This problem can be loosely compared to the problem in NOMIS of authentication to use particular datasets which was simple with MTS file permissions but is more restricted on the Unix system with only three levels of file permissions.

STYLE is written with a combination of Fortran and C modules which need to be linked. There are compiler differences in the way C routines can be called from Fortran routines so this also had to be considered when porting the code.

It should be noted that the proportion of portability problems arising from Programming Language criteria in the porting of STYLE amount to 0.07% of the total lines of code. Portability problems as a whole (including operating system problems, machine architecture problems etc.) amount to 2.73% of the total lines of code so it can be seen that the criteria obtained from programming language problems are only a small proportion of the criteria affecting portability.

Operating System	Sun (SunOS4)	Fujitsu $\sum$ ($\sum$ OS)	IBM 5570 (OS/2)
High Level Modules (Fortran)	95.3	95.4	95.0
Low Level Modules (C)	60.9	59.7	40.0
Total	93.8	93.7	91.5

Fig 2.6a: Mobility of STYLE modules.

Using Tanaka's formula for obtaining a measure of the mobility/portability of STYLE the results in Fig 2.6a are obtained when STYLE is moved to three different operating systems:

The second stage of the experiment was to convert all the Fortran Modules to C mechanically (with some manual modification to improve performance). The table in Fig 2.6b shows the results of applying the mobility function to the C-converted version of STYLE for the same operating systems:

Machine	Mobility
Sun	99.9
Fujitsu $\sum$	99.8
IBM 5570	91.8

Fig 2.6b: Mobility of C-converted STYLE program.

As can be seen the mobility is improved in all cases by re-writing the Fortran modules in C. Tanaka attributes this improvement to the elimination of the variants of the Fortran compiler. It is not clear whether each of these machines had the same C compiler and it should be emphasised again that the C language definition itself has variants so the problems of Fortran may not always be solved by re-writing in C.

Tanaka discusses the use of a Master Source File, exported over a network as a means of maintaining distributed software. This was done for the STYLE program with the master source file being held on a Sun 4 machine. It was found that the workload for source maintenance and porting was reduced by 66% and the disk space required was reduced to $\frac{1}{n}$ (where n is the number of machines on the network) which, in the case of STYLE, was a reduction of 15MB.

Another example of porting an existing program is given by Lemoine and Mullor [LEMO81]. Their paper is based on a number of experiments in transferring GERMINAL which is a general CAD System written specifically for a CII-IRIS 80 machine to

machines such as an IBM 360/370, a UNIVAC 1110 and a PDP 10. The program has not been designed with portability in mind and although the solution is very different there are similarities between this problem and that of the NOMIS program considered in Chapter 6. Both problems are basically the transfer of a complete system which has been designed very specifically to run on a particular system.

The GERMINAL system was written entirely in LP70, a language very close to the IRIS 80 Assembler but with a syntax including some high level language constructs such as IF THEN ELSE and loops. GERMINAL is an experimental tool and so its 30000 lines of code were not written to include portability characteristics. Some of its important characteristics are:

(i) The definition and manipulation of data structures are highly IRIS 80 system dependent.

(ii) The system is segmented into functional modules of varying sizes ranging from a few dozen statements to several thousand.

(iii) Data communication between modules is carried out through common global zones and registers which increases the interdependence of modules.

(iv) Calls to the IRIS 80 operating system are strictly localised to a few modules. This is a positive feature when considering the portability of the system.

Lemoine and Mullor list a number of possible methods of porting the GERMINAL system together with their reasons for rejecting each method:

(i) Re-writing the system in a high-level language - The language chosen would need to exist on all possible target machines and be "highly normalised" so would by necessity need to be a well-standardised language. The major disadvantages with these are that they are not adapted to the problem of system writing and they are insufficient in dealing with data structures such as bits, bit groups and stacks. It was therefore expected that the use of this method would require a complete re-think of the data structures and such a re-write was estimated at 5 man/years minimum resulting in a less efficient system.

(ii) The use of macrogenerators - This was found to be limited by the existing tools which would require very delicate debugging; an ineffective analysis of the original text and the need to modify it manually; an expansion rate of between two and five for even the smaller modules (100 statements). Thus the prohibitive factors of

macrogenerators were the large-scale manual modification and the highly inefficient expansion rate of the code.

(iii) Compiler Transfer - This can be further divided into two methods, write a compiler for each machine considered or modify the existing compiler to produce code which can be used by each of the machines considered. The first solution of re-writing the compiler has a more general use since it would permit a new language to be made available on the target machines and enable all software written in that language to be ported to those machines without change. However, it is a long and costly job to write a compiler and it would be necessary to write as many compilers as there were target machines. The second method, while easier to implement, assumes a perfect understanding of the existing compiler and that there will be a similarity between the machines under consideration as this solution does not consider semantic problems linked to the source machine.

The actual method chosen for the porting of the GERMINAL system was the use of a general language translator to convert the LP 70 code to the target machine language. The advantages of this method were:

(i) The simplicity of the grammatical representation of the source language.

(ii) The power and diversity of the rules for translation into the target language.

(iii) The use of a conversational precompiler which is independent of the source and target languages and has debugging tools. The actual tools used were LDSS (Syntax and Semantic Descriptive Language - English translation) and PRECOMP (PRECOMPilateur Universel - Universal Precompiler).

Lemoine and Mullor give details of two specific case studies. The first is the transfer of GERMINAL to IBM 360/370 machines. The code is given two automatic passes with translators written in LDSS followed by a manual pass to solve any remaining problems. These include restructuring the code and data segments to comply with IBM addressing constraints, further optimisation of the code generated and re-writing the I/O modules with an assembler. The duration of this project was 2 man/years.

The second case study was the transfer of GERMINAL to UNIVAC 1110 and PDP 10 machines. The LP 70 code was descended gradually to the target machine assemblers through several layers of abstract machines. This was done in two passes with the aid of translators written in LDSS. The third phase, specific to the target machine, was a

transition to the real assembler of each computer. The automation of this chain was more developed than in the IBM 360/370 example since the problems of conversion were not so extensive. The manual intervention was much reduced resulting in the duration of this project being only 18 man/months.

2.6.2. Designing for Portability

A good example of software designed with portability in mind rather than the porting of an existing program is given by Hatton et al. [HATT88]. Their paper describes the design and implementation of a 500000+ line portable Fortran 77 package for the processing of Seismic data, the Seismic Kernel System ((SKS). The implementation of SKS contains two kernels which are written for the specific implementation. Originally there was just one - the "device-dependent" kernel but, for increased efficiency, a second kernel - the "activity-dependent" kernel was identified to contain all the routines which contribute significantly to the CPU and I/O overhead of the program.

The device-dependent kernel comprises about 0.5% of the code and the activity dependent kernel about 1%. Thus the remaining 98.5% of the code is identical on all machines. It is important to note that these kernels are almost static and as the application grows it is the 98.5% of machine-independent code which is increasing in size.

Using the formula for mobility defined by Hatton et al., and given in Section 2.4 SKS is shown to be more than 99.8% portable, this value being higher than any noted for the Fortran version of the STYLE program discussed earlier.

The SKS package was originally designed on a Sperry Univac 1100/62 with a particularly good Fortran compiler. After porting to a Digital VAX 11/780, some portability deficiencies were found and the package was re-engineered over a period of 6 man/months. In 1984 the then 300000 line package was ported to a Cray IS in 4 weeks by one person. Since then it has been successfully ported to Amdahl, CDC, Convex, Data General, DECC, FPS, Fujitsu, Honeywell, IBM, Perkin Elmer and Prime machines. On average the device-dependent kernel takes around 4 person/weeks to produce for a machine and it is noted that the time taken is directly proportional to the quality of the Fortran Compiler on the target machine.

More recently SKS development has been moved to Unix systems. A particularly useful tool, Flint, has been brought in to help with maintaining of programming standards throughout the code. Flint is a very sophisticated source code verifier which strictly

enforces the use of a portable subset of Fortran. This automatic enforcement is considered very important by Hatton et al., particularly in the case where large numbers of programmers are working on the same program.

Wallis [WALL82] gives an example where, in some cases, care should be taken in designing with portability in mind. A large Spacial Data Processing program written at the University of Bath is considered. The maximum size of an integer is normally at least 2^{15} on 16-bit computers and is much larger than this on many mainframe machines. A maximum integer size of 2^{15} was considered inconveniently small so 2^{23} (the next smallest common maximum) was more adequate for this program. The use of such a value allowed the program to be ported to most mainframe machines but not to microcomputers. In this case, it was unlikely that the program would ever be required to be ported to a microcomputer so the restriction in the range of potential target machines was accepted in exchange for easing the constraints imposed by the requirements of portability.

2.7. Summary

It can be seen from the definitions given in this chapter that portability does not have one single definition and there is certainly not one standard measure of the portability of a program. As Mooney [MOON90] states, portability is very much a "matter of degree."

It can therefore be concluded that a measure of portability can only be made if the set of target systems is restricted to a finite (if large in some cases) number. In this case some measure can be made and formulae defined for the portability of programs.

A large number of criteria affecting portability have been listed, both general criteria and those specific to particular programming languages and systems. The criteria affecting the portability of Fortran are of most interest to this thesis and it will be seen that many of the criteria listed in these papers are those included in the following chapter where the criteria affecting the portability of Fortran programs from a particular system are considered.

The formulae given here for measuring portability (or mobility) provide a starting point for defining a more detailed Portability Function which takes the effects of individual criteria into account. The examples given, particularly the SKS system described by Hatton et al., [HATT88] show that, by taking heed of the recommendations, software can be produced which is both efficient and portable giving it an increased lifetime since it can be ported with the minimum of effort to new systems as they are developed.

Chapter 3. Portability Criteria

3.1. Introduction

This chapter contains a detailed description of some of the important criteria affecting the portability of Fortran 77 programs. The list does not include every criterion that could possibly affect Fortran 77 programs (it would be impossible to list every such possible criterion) but includes those criteria that were considered most important after evaluating the portability of Fortran programs in the literature. In particular the criteria were tailored for Fortran programs written for the MTS operating system. In some cases these criteria are constructs left over from a previous Fortran 66 compiler, in other cases they are due to IBM Fortran 77 language extensions. The use of system specific subroutines and libraries is also considered.

3.2. Discussion of Criteria Affecting Portability

The following is a list of the criteria found to affect the portability of Fortran programs. They are in no particular order here, although they are grouped together in the summary at the end of the chapter.

3.2.1. Execution of 'DO' Loops

This criterion is dependent upon the particular compiler (rather than the machine) and is a specific difference between Fortran 66 and Fortran 77 Compilers. In Fortran 66 a DO loop is executed at least once whereas in Fortran 77 this is not, in general, true. Depending on the particular expression, a DO loop may not be executed at all. The following example illustrates this criterion.

Take the following section of Fortran Code:

```
      N = 0
      ISUM = 0
      DO 10 I = 1, N
        ISUM = ISUM + 1
   10 CONTINUE
```

This section of code will compile with both Fortran 66 and Fortran 77 compilers however its run-time behaviour will be completely different. In the Fortran 66 case the DO loop must be traversed at least once and so, despite the expression in the DO loop giving (with N = 0) I = 1, 0, the DO loop will be traversed once leaving ISUM = 1

after the loop. In the Fortran 77 case the DO loop will not be traversed at all since the condition cannot be satisfied so ISUM = 0 after the loop. As can be seen this could have potentially dire consequences if a program containing such a DO loop was ported to another system and compiled with a compiler not operating in the same way.

3.2.2. Free Format Source Statements

The Fortran 77 standard does not allow Free Format Source statements. That is statements are restricted to columns 7 - 72 with column 6 reserved for the continuation facility and columns 1-5 reserved for statement labels. Many compilers (in particular Fortran 66) allow Free Format source statements where these restrictions do not exist, however for true portability this must be considered a criterion affecting portability.

Another criterion linked to the format of source statements is that many modern compilers (such as Sun Fortran) allow the source code to be in mixed or lower case. This is not portable since other compilers insist on upper case source code. Often the reverse of this criterion can occur. Source code may be ported from an older machine which insists on upper case source code to a machine with a more modern compiler, allowing mixed case. In this situation the code could potentially be converted to lower case. This would have the result that an originally portable piece of code is ported, converted and becomes a non-portable piece of code!

The case of filenames referenced from within the code is also important. On some systems (notably MTS) the case of filenames does not matter, however other systems (such as unix) may have case sensitive filenames so any referenced from within the code must be in the correct case whether the code is written in free format or not.

3.2.3. Format of Variable Names

Some compilers still allow $ as a valid character in variable names. They also make the assumption, in the absence of a variable declaration, that if the first symbol of the variable name is a $ then the variable is a REAL of length 4. This is non-standard Fortran 77 and should be avoided.

Many modern compilers allow long variable names but standard Fortran 77 indicates that only the first 6 characters of a variable name will be considered internally. For example FLNAME1 and FLNAME2 would not be considered unique whereas FNAME1 and FNAME2 would be considered unique. Some compilers refuse to allow longer

variable names whereas others ignore the excess characters so for general portability a maximum of 6 characters should be used for variable names.

3.2.4. Variable and Function Length Specifications

Although considered as separate criteria these two can be discussed together since the same restrictions apply to both function and variable specification statements. The Fortran 77 standard does not allow length specifications in declaration statements (for functions and variables) for data types other than character. Originally these length specifications for other data types were IBM extensions but they are now available with most compilers. This criterion emphasises one of the problems of deciding on portability criteria. Obviously this has to be included as a portability criterion since it violates the Fortran 77 standard, however, since most compilers allow length specifications how important is this criterion? If "General Portability" is being considered then this criterion has to be as important as any other. If "Specific Portability" is being considered then, in many cases this criterion will have no effect at all and need only be given a "Portability Score" if the machine the code is being ported to does not have the extension.

Some examples of non-portable length specifications are given in Fig 3.2a with some of their possible replacements:

Non-Portable Construct	Standard Replacement (if available)
REAL*8	DOUBLE PRECISION
INTEGER*2	
LOGICAL*1	CHARACTER*1
COMPLEX*16	
DOUBLE PRECISION COMPLEX	

Fig 3.2a: Non-portable length specifications.

The latter two examples are equivalent. COMPLEX*16 can be replaced by DOUBLE PRECISION COMPLEX, however this, itself, is an extension accepted by most compilers, although not in the ANSI Standard.

3.2.5. Data Initialisation

Some compilers allow data initialisation in type specification statements but this is not standard and DATA statements should be used instead. For example:

```
REAL PI
INTEGER FIRST
DATA PI/3.142/
DATA FIRST/1/
```

Would satisfy the standard whereas:

```
REAL PI/3.142/
INTEGER FIRST/1/
```

would not and would therefore be considered non-portable.

3.2.6. Order of Statements

The Fortran 77 standard has a recommended order for statements. The table in Fig 3.2b gives the recommended order of all statements in the source code, together with possible overlap as set down by the Fortran standard. For true "General Portability" every statement in the code should conform exactly to this table.

<table>
<tr><td rowspan="6">Comment lines</td><td colspan="3">PROGRAM, FUNCTION, SUBROUTINE, BLOCK DATA</td></tr>
<tr><td rowspan="5">FORMAT, ENTRY</td><td rowspan="2">PARAMETER</td><td>IMPLICIT</td></tr>
<tr><td>Other specification statements (REAL, DIMENSION, etc)</td></tr>
<tr><td colspan="2">DATA</td></tr>
<tr><td colspan="2">Statement function</td></tr>
<tr><td colspan="2">Executable statements</td></tr>
<tr><td colspan="4">END</td></tr>
</table>

Fig 3.2b: Order of Fortran Statements.

Checking the order of every statement would be beyond the scope of the simple portability function considered here so we take as the criterion, a more common ordering problem which is another difference between Fortran 66 and Fortran 77 compilers which are less flexible about ordering. Fortran 66 is fairly flexible over where DATA statements may occur so the following would be allowed:

```
DATA TEST/.TRUE./
LOGICAL TEST
```

If compiled with a Fortran 77 compiler this would fail since DATA statements must come after type declaration statements. The following would be the correct order for these two statements and would compile, independently of the compiler used:

```
LOGICAL TEST
DATA TEST/.TRUE./
```

3.2.7. Hollerith Strings

Hollerith Strings used in FORMAT statements are another criterion left over from Fortran 66 compilers. They were necessary before character string manipulation was available but have no use in Fortran 77 and many compilers do not recognise them. A Hollerith string was always given a length and was defined as follows:

Say a FORMAT statement was required to print out a person's name. This would be done as follows:

```
FORMAT (9HLIZ GANDY)
```

This can be replaced by the Fortran 77 statement:

```
FORMAT ('LIZ GANDY')
```

which is portable and independent of the compiler used.

3.2.8. Alternative Subroutine Return Calls

Subroutine calls may contain extra parameters which are statement numbers of the next statement to be executed depending on the return value of the subroutine. In standard Fortran 77 these statement numbers are indicated by a *. For Example:

```
        CALL SUB (A,B,*99,*100)
        CONTINUE WITH PROGRAM
        .......
   99   CALL ERR1(A,B)
        GOTO 1000
  100   CALL ERR2(A,B)
 1000   STOP
        END
```

In this example, if the return value from the subroutine SUB is 1 then statement 99 is executed next, if the return value is 2 then statement 100 is executed next, otherwise the program continues as normal.

In IBM Fortran or Fortran 66 then the & character may be used instead of the * in the subroutine call. Thus the above subroutine could be:

```
CALL SUB (A,B,&99,&100)
```

This construct will prevent the program compiling with other compilers but is a fairly simple criterion to detect and correct to increase portability.

3.2.9. Character Handling

In the days of Fortran 66 there were virtually no character handling facilities in the language which was why Hollerith strings were used. Since then character handling has become much more advanced, however there are a number of points and criteria that should be noted where many compilers differ from standard Fortran 77, particularly IBM Fortran compilers. The standard has two types of storage unit, one for numerical values and one for characters. These remain separate giving rise to the following criteria:

3.2.9.1. Common Blocks

The Fortran Standard states that if there is a character variable in a COMMON block then all variables in that COMMON block must be of character type. Other compilers may allow character variables to be declared in the same COMMON block as other data types such as integers or reals.

3.2.9.2. Equivalence Statements

A similar criterion to that above is that of restrictions to EQUIVALENCE statements. The Fortran 77 Standard states that character variables may only be equivalenced to other character variables.

Many compilers allow character variables to be equivalenced to other data types and this is widely used, particularly equivalencing character variables with integer variables. However, again this is a non-standard feature and it must be considered when investigating the general portability of a Fortran program.

3.2.10. System Specific Subroutine Calls

By definition these are the most common cause of concern when porting code from one system to another. System specific subroutines can be defined as any routine called by the Fortran code which is specific to the machine or operating system being used.

They can be split into a number of different categories, all of which must be considered when examining the portability of the code.

3.2.10.1. Character Handling Routines

Some systems have system specific routines performing character handling operations. Many of these can be re-written in standard Fortran 77 which can then be ported to the new system either with the application code or as more widely available routines for the use of more general applications on the new system.

3.2.10.2. Program Environment Subroutines

These are subroutines which are specific to the operating system and provide information about the program environment. Some examples are:

user identifier, loginame etc.

date and time (as character strings or integer values).

system accounting routines e.g. elapsed time, cpu time for job.

The information provided by these routines is usually available on most systems, however the methods of calling these routines may differ. So, while major changes to the code may not be necessary before porting the code, these program environment subroutine calls must be detected and the new operating system checked for their existence and appropriate changes to the subroutine calls made.

If the routine required does not exist on the new machine then large amounts of code writing and re-writing may need to be done.

3.2.10.3. Bitwise Logical Operations

Many systems provide system specific subroutines which perform bitwise logical operations. As with program environment subroutines, most systems provide some facility for performing these operations, however the method of calling these routines, parameters required and return values may differ between systems.

3.2.10.4. Special Library Routines

Many systems have special libraries of routines available to be called by Fortran subroutine calls. One such example of a special library of routines is the NAG Library. Any Fortran code calling these routines is portable to another system provided the special library is available on the new system.

3.2.11. Internally Defined Filenames

This is a portability criterion more specific to IBM Fortran and the VS Fortran Compiler (that which was available on the MTS system). On such systems, defining filenames internally with OPEN statements is limited to those filenames containing no more than 7 characters of which the first is a letter and only letters and digits are allowed in the remainder of the name. Therefore the following statement:

```
OPEN (UNIT=2, FILE='INTFILE')
```

would be allowed whereas:

```
OPEN (UNIT=2, FILE='-INTFILE')
```

or

```
OPEN (UNIT=2, FILE='CLOO:INTFILE')
```

would not be legal. Both -INTFILE and CLOO:INTFILE are legal filenames on the MTS system but their use within the OPEN statement would be illegal. To get round the problem programmers for the NOMIS system have avoided the necessity of using variable substitution such as the following:

```
CHARACTER*12 FLNAME
DATA FLNAME/'CLOO:INTFILE'/
.......
OPEN (UNIT=2, FILE=FLNAME)
```

where an extra variable needs to be declared, by assigning the file using the FTNCMD system specific subroutine as follows:

```
CALL FTNCMD ('ASSIGN 2=CLOO:INTFILE', 21)
```

where 21 is the length of the string and the FTNCMD subroutine performs the action of the string in the operating system.

The use of the FTNCMD subroutine to assign filenames is widely used in MTS but problems occur when the code is transferred to non-IBM systems. The limits on the length and characters in the filenames may not exist so that a straight OPEN statement

may be used, however changes to the code will be necessary to replace calls to FTNCMD with OPEN statements.

Another problem related to this is that filenames which are legal on one system may be illegal on another. If these filenames are defined internally then the code will need modifying if these files are ported to a new system and renamed.

3.2.12. Units Assigned at Runtime

The ability to allow files to be assigned to units at runtime is another widely used but non-portable IBM extension. Units can be referenced in the code without being assigned to a particular file. At runtime these can be assigned from the command line to allow variable data files, say, to be used. For example if the object code for a particular program is PROG and units 21 and 22 are referenced in the code but not assigned to particular files then the following lines of code would be valid ways of running the program and assigning the data files:

```
# RUN PROG 21=DATA1 22=RESULTS1
# RUN PROG 22=RESULTS2 21=DATA2
```

If this code were transferred to a non-IBM system there may be no way of assigning unit numbers to filenames at runtime and therefore changes to the code would be necessary.

3.2.13. Character Codes

One portability criterion arises from the difference between systems running with EBCDIC character codes and ASCII character codes. There are usually routines for converting between EBCDIC and ASCII so moving data files between two differing machines should not cause problems, the only overhead being the necessity to process each data file with the appropriate conversion routine and often this is done transparently during file transfer. The problems occur when the actual source code makes references to the actual EBCDIC or ASCII code numbers rather than the character itself. For example:

```
IF X.EQ.64 (EBCDIC) or IF X.EQ.32 (ASCII) rather than IF CHAR(X).EQ.' '
```

use problems moving between two machines with different character sets.

Another problem associated with this one is that data files containing both text and binary may be difficult to transfer between machines with different character codes. Take,

for example, a file containing records of binary data with text headers being ported from a machine with EBCDIC codes set to a machine with ASCII character codes. Ideally the binary data should remain unchanged and the text should be converted from EBCDIC to ASCII. If a binary transfer is used to move the file then the text will not be converted to ASCII, if a text transfer is used then the binary data will be considered as text and converted to ASCII as well,making it useless.

If this situation occurs then it would be necessary to transfer the files as binary and convert the text to ASCII later which could prove complicated and would involve writing extra code.

3.2.14. Quote Symbols

It is possible to run Fortran code on some machines where a text delimiter of double quotes is allowed as well as the standard single character text delimiter. It is interesting to note that this criterion was detected as a difference between two unix machines the GOULD NP1 which allowed both double quotes and single quotes and the SUN4 which allowed only single quotes. For portability the use of double quotes should be avoided.

3.2.15. Default-Unit Numbers

Many Fortran programs rely on default units being assigned to standard input and standard output (and often standard error as well). Whilst most systems use unit 5 for standard input and unit 6 for standard output, some use 1 and 2 respectively instead and others may differ again or have no defaults set. There is no standard set of defaults so it is necessary to check before porting code containing references to these default unit numbers whether the same defaults exist on the new system.

3.2.16. Statement Labels Limit

The number of statement labels in a particular program subroutine is subject to an upper limit and this upper limit is dependent on the particular machine or compiler being used. Therefore code which may work on one machine may not compile on another if the maximum number of statement labels allowed per subroutine is lower and this code contains a subroutine where the number of statement labels exceeds this maximum value.

It is therefore necessary when porting code to check that the limit on the new machine is not lower than on the current machine and if it is to subsequently check that no subroutine in the code has greater than this value of statement labels.

3.2.17. Limit to the Number of Concurrent Units Open

At runtime there is a limit to the number of units allowed to be opened concurrently. As with the statement labels above this limit is dependent on the machine being used. The difference here is that this limit is for the program as a whole and is detected at runtime rather than being a limit for each subroutine and being detected when the code is compiled.

3.2.18. Hexadecimal Data in DATA Statements

This is a language extension whereby the Z code in a DATA statement is used to indicate that the variable is to contain hexadecimal data. The use of hexadecimal data, initialised in DATA statements, is non-standard and detrimental to portability.

3.2.19. Z and Q Format Types

Whilst there are a large number of standard format types used in FORMAT statements, for example I, A, and F format types, there are a number of extra format types available as extensions to the Fortran 77 standard. Two IBM extensions are the Z format code to indicate hexadecimal data and the Q format code to indicate extended precision.

3.3. Summary

Having considered some of the criteria affecting the portability of Fortran it is now possible to group these criteria. They naturally fall into three categories, criteria concerned with the Operating System used; criteria affecting the format of the source code; and criteria which are extensions to the Fortran Language Standard. A summary of the criteria described in this chapter is given below with the criteria grouped according to these three criteria types.

Operating System Criteria:

Default unit numbers.

Concurrent unit limits.

Internally defined filenames.

System specific subroutine calls.

Units assigned at runtime.

Limit to the number of concurrent units open.

ASCII/EBCDIC character codes.

Format of Source Code Criteria:

Order of statements.

Quote symbols.

Free format source statements.

Case of source code statements.

$ in variable names.

Alternative subroutine return calls.

Language Extensions:

Character variables in EQUIVALENCE statements.

Character variables in COMMON blocks.

Variable length specifications.

Function length specifications.

Length of variable names.

Hexadecimal data in DATA statements.

Z and Q format types.

Hollerith strings.

Data initialisation.

Execution of DO loops.

Statement labels limit.

Chapter 4. Portability Measures

4.1. Introduction

After the theoretical discussion of the previous chapter, which identifies the criteria which need to be considered when porting a Fortran program from one system to another, this chapter considers how the occurrence and effects of these criteria can be measured. Firstly, there is a discussion of how various tools and programs can be used to identify the occurrence of criteria in the code. After a number of specific examples are demonstrated, a portability function is defined whereby the measures obtained can be weighted according to their effect on portability and grouped together to give an overall measure of the percentage portability of the code so that comparisons can be made.

Finally, there is a detailed discussion of how particular criteria can be measured and the weighting factors given to these criteria, taking the NOMIS example described in Chapter 6, of moving from the MTS system to Unix, as a basis for these figures.

4.2. Tool Requirements

Before considering the actual measures and scores given to each criterion, it should be determined what tools and programs are available on both systems which might aid the detection of each criterion. The objective is to provide an assistant for porting Fortran code in terms of measures and criteria, rather than to write full static analysis tools so it is important to make as much use of available tools as possible.

Obviously, these tools themselves will be specific to the systems considered, they may be on the original system, in fact this will be necessary for any tools which work on a compiled version of the code since at the point where portability is considered, the code will probably not compile on the destination system. Other tools, not relying on compiled versions of the code, may be available on the target machine or tools may even be found on other machines.

The compiler itself may be used to obtain measures of some criteria, particularly if, for example, an option is available to flag statements not conforming to the Fortran ANSI standard. Obviously, the extent of the criteria measures obtained from the compiler warning messages will vary with different compilers.

Both the original and destination systems should also be looked at for the existence of other tools which may be made use of such as those which may produce lists of external

and system specific references and any programs which may tidy the source code, making it more portable.

There will always be some criteria which will need to be searched for directly in the source code so the existence of a pattern matching and processing tool such as grep or awk can be invaluable in searching through large amounts of source code for particular statements. For example the occurrence of DO loops containing expressions can be detected using a simple awk script. These pattern matching tools can also be used on the output from other tools to obtain the actual measures of criteria in a form which is easier manipulate when applying the portability function.

By making use of existing tools and programs a lot of effort can be reduced in obtaining measures of the criteria affecting a particular piece of code.

4.3. Examples of Specific Tools

Having considered the general types of tools that may be available to help in obtaining measures of portability criteria this section now considers a number of examples of such tools, available on either the MTS system or the Sun 4 Unix system used in the NOMIS example described in Chapter 6. These tools are described in more detail below:

4.3.1. Compiler Warning Messages

The simplest way of detecting and counting some of the criteria affecting portability in a piece of source code is to use the compiler's warning messages and search for the message text appropriate to that criterion. The warning message for each criterion will be a unique text string so a simple awk script can be written to detect these specific text strings from a log of the compiler warning messages.

In the examples considered the source code is being moved from the MTS system using a VS Fortran compiler to a Sun 4 Unix system using a Fortran 77 compiler (f77). In order to detect these criteria from the warning messages it is necessary that the code will compile under that compiler. In this case we consider the warning messages produced by the VS Fortran compiler on the MTS machine. This compiler contains a Source Language Flagger feature which will flag Fortran Statements that do not conform to the syntax of ANSI Standard Fortran 77.

It should be noted, however, that this option for flagging statements is only available for those statements not conforming to the Fortran 77 standard, it is not available for

flagging statements which do not conform to the Fortran 66 standard. It is also not possible to use the source language flagger for code written in free format.

The following criteria are picked up by the VS Fortran Source Language Flagger. (These items are also considered in more detail in Section 4.5). Note: not all of these features are considered relevant for the criteria:

4.3.1.1. Global Items Flagged

Continuation statements where columns 1 to 5 are not blank.

The $ symbol used in a name.

Non-character variables where an actual length is specified.

Explicit type specification statements for REAL*16, COMPLEX*16, COMPLEX*32.

Hollerith strings in statements other than FORMAT statements.

Hexadecimal constants used as data initialisation .

4.3.1.2. Statements Flagged

A large number of statements which do not conform to ANSI such as AT, EJECT, NAMELIST, INCLUDE.

COMMON statements where character and non-character data are in the same block.

DATA statements where the statement appears before the end of the specification statements or Q, Z or Hollerith constants are used.

EQUIVALENCE statements where character and non-character data are equivalenced.

FORMAT statements using Q and Z format codes.

FUNCTION statements where a length is specified for a real, logical, integer or complex function.

IMPLICIT statements where a length is specified for a real, logical, integer or complex range or the $ symbol is used as an alphabetic character.

INTEGER, REAL, COMPLEX and LOGICAL type statements where data initialisation is specified.

4.3.2. FTNTIDY Program

For those criteria not picked up by the VS Fortran Compiler warning messages, it is necessary to find other methods of counting the number of occurrences of particular criterion. The FTNTIDY program on MTS was used in obtaining this information for a number of criteria considered.

FTNTIDY can be used for two purposes, to tidy the Fortran source or to produce cross reference dictionaries of variables, statement labels, functions, subroutines and logical I/O units used.

Both of these features can be used for obtaining counts of particular criteria. Many tidying features are available within FTNTIDY, in particular the removal of all blanks, except those in Hollerith strings or text strings, and the insertion of single blanks to improve readability. The advantage of this is that when using awk scripts on a piece of tidied source code it can be guaranteed that the spaces will be in such a way as to distinguish exactly the format of particular commands.

The main use of FTNTIDY, however, is in the production of four cross-reference dictionaries which can be examined to obtain valuable information about certain criteria. The following four dictionaries are produced:

(i) Subprograms, which include all subroutines, functions and entries.

(ii) A list of variables and their types indicated in a coded format so it is very easy to use an awk script to detect the number of variables of a particular type.

(ii) A list of all statement labels accessed in the code. From this it is easy to detect the number of statement labels in the code.

(iv) A list of logical I/O units referenced within the code. From this can be obtained the total number of I/O units referenced and, of equal importance, which I/O units (including the default ones) are referenced. It is worth noting, however, that any measure taken using this dictionary may be an under-estimation since this dictionary does not include logical I/O units referenced by variables. Therefore, any counts obtained from this dictionary should be considered an approximation only.

The FTNTIDY program has been used extensively, together with awk scripts to search the dictionaries, in obtaining measures of particular criteria in the NOMIS example. A sample of the output produced by FTNTIDY can be seen in APPENDIX 1.

4.3.3. EXTREFS Program

Another tool is the EXTREFS program on MTS which allows lists of all external references made by the program to be obtained. This program differs from those previously mentioned in that it takes as its input the compiled version of the code (the object code).

The EXTREFS program provides three lists:

(i) A sorted list of program subroutines and functions. This list contains the subroutine or main program itself and any other subroutines or functions compiled within the same program.

(ii) A sorted list of unresolved external references. This list contains all references made by the program that are not compiled within the same program. This will include all subroutines compiled separately be they Fortran routines, other language routines such as Assembler or routines belonging to specialised library packages such as NAG. In most cases, if a whole set of routines are being ported, then these external references do not adversely affect the portability since they will simply be ported with the program. The exception to this rule would be when the routines were those belonging to external libraries such as NAG which would affect portability if that particular library was not available on the destination machine. The best way to separate these library routines would be to run EXTREFS on the program compiled as a whole so that all external references linked to other subroutines of the same program were eliminated. Those left would include routines belonging to such libraries and it would then be necessary to consider whether these libraries were available on the destination machine.

(iii) A sorted list of MTS system specific subroutines referenced. This is the most useful piece of information made available by the EXTREFS program. MTS system subroutines are the system specific subroutines which, as criteria themselves, have a major effect on the portability of a program.

As with the other tools the output produced by EXTREFS can be logged and awk scripts written to pick out the relevant criteria.

4.3.4. Awk

Awk is a pattern scanning and processing language available under Unix that can be used in this work as a tool for obtaining the measures of the occurrences of particular criteria. Awk scripts can be written to search the information provided by the tools and programs already considered such as the tidied source code and cross-reference dictionaries produced by FTNTIDY, the external references index produced by EXTREFS and the log of compiler warning messages. Scripts can also be written to search the raw source code for criteria not picked up by these other tools. An example of such an awk script is given in Fig 4.3a.

```
# Count number of statement labels
# Input log of FTNTIDY from MTS
# Output variable.list

BEGIN {count=0}

$2~/^STATEMENT$/ && $3~/^LABEL$/ && $4~/^DICTIONARY$/ {
        getline
        getline
        getline
        while (($1 != "***")&&(substr($0,3,6) != "TYPES:")) {
                if (substr($0,4,6) != "      ") count++
                getline
        }
}

END { print "Number of statement labels: " count }
```

Fig4.3a: Example of an Awk script.

This awk script is used to search the cross-reference dictionary produced by FTNTIDY to obtain a count of the number of statement labels in the program.

Awk was used extensively for obtaining measures of criteria affecting the portability of the NOMIS example.

4.4. Portability Function

4.4.1. Portability Score

The portability of a particular program or routine has been calculated using a simple weighted function. Each criterion identified has been given a particular score or

'weighting factor' These scores are in the range 0-10 with the system specific routines covering the whole range, whereas the other criteria, which are not in general as serious, only cover the range 0-5. These weighting factors are in ascending order of seriousness with 0 meaning the criterion would have no effect on portability. This should never occur since a criterion with no effect on portability is not a criterion at all. However if the system is being taylored for a particular application and one of the criterion is not relevant in this case, the weighting factor can be set to 0 so any occurrences of this criterion will be ignored. The weighting factors increase to 10 which means the criterion would cause complete non-portability of the program. That is, there is no way the program could even be re-written to enable it to work on another system. This should rarely occur since there are very few programs which cannot be re-designed to work on another system. Potentially this weighting factor could only be given to a system specific subroutine which performed some major task only relevant to that particular system. If this routine was required on a different system then it would normally be expected that a similar routine would either exist or could be written otherwise there would be no point porting the program in the first place.

All the criteria considered here have been given a weighting factor ranging from 1, very simple correction, to 9, a re-write of either part of the program or a whole system specific subroutine called by the program. A discussion of the reasoning behind the chosen weighting factor for each of the criteria is given in Section 4.5.

From these individual weighting factors and a count for the occurrence of the criterion in the code a 'Score' can be obtained for the program using the following summation:

$$Score = \sum_{i=1}^{n} x_i w_i$$

where

x_i = count for criterion i for this program.

w_i = weighting factor for criterion i.

This is summed over all criteria considered. The count x_i, for each criterion, is the value determined from analysis of the code. It's value is dependent on the particular criterion and can be the number of occurrences of a criterion in the code. For example in the case of ASCII/EBCDIC violations the count is the number of possible occurrences

of this criterion in the program. In this case the count can be quite high and so the weighting factor for this criterion is quite low (namely 2) so that it does not dominate the overall score because there may be a large number of these and some (or many) may be spurious. Alternatively, the count can simply be the occurrence of a particular criterion, independent of the number of occurrences of that criterion. For example, for variable length specifications the count would be the number of variables declared with non-standard length specifications, no matter how many references to these variables there are in the code.

In the case of system specific subroutines the score is further sub-divided into separate routines and their individual scores which may vary according to the weighting for that routine. In this case the System Specific Score can be defined as:

$$Machine\ Specific\ Score = \sum_{j=1}^{k} s_j m_j$$

summed over all system specific subroutines where

j = each individual subroutine

s_j = Occurrence of System Specific Subroutine i.e. $s_j = 0$ if the routine is not called in the code. $s_j = 1$ if the routine is called. $s_j = 1$ no matter how many calls to the routine in the program, the reasoning behind this being that once a replacement or modification has been decided for the first call, it will be similar for each other call.

m_j is the weighting or score for each individual system specific subroutine (in the range 0-10 ranging from totally portable to totally non-portable).

For the number of statement labels and unit numbers referenced the portability is not affected unless the number of statements or units referenced is greater than a pre-defined maximum value. In this case the score:

$x_i w_i = (x_i - Max_i)\ w_i$

Where Max_i is the maximum allowed.

That is, the number greater than the maximum limit multiplied by the weighting factor for that criterion. The maximum values are built into the code for the reporting routines and can be easily changed to correspond to the limits for the particular system the program is to be ported to. In the case of porting to a Sun 4 the maximum number

of statement labels allowed is 401 for each subroutine. The maximum number of units which can be referenced is 63 in any one complete program.

These separate components can be added together to give a total score for the program. This can be defined as follows:

$$Total\ Score = \sum_{i=1}^{n} x_i w_i + \sum_{j=1}^{k} s_j m_j$$

The total score given to the program as a whole (i.e. taking all the component subroutines into consideration) is not simply the sum of all the individual routines. Without the criterion of the maximum number of unit numbers this would be the case. The individual scores for each criteria can be summed over all subroutines. In the case of the maximum number of units referenced the problem occurs at runtime rather than during compilation. Therefore each individual subroutine could compile without error if they each had less than the maximum number of units allowed, but over the whole program there could be more than the maximum allowed. For the total score for the whole program therefore, the sum of the units for each subroutine is taken as x_i and the difference between this and the maximum value is multiplied by the weighting factor for this criterion.

This same problem does not occur for the other criterion dependent on a maximum value - the number of statement labels as this maximum is for each individual subroutine i.e. two subroutines could have 400 and 300 statement labels respectively, causing 0 score for this criterion (as the maximum is 401) and although there would be 700 statements labels overall, the score would still be 0.

In the case of unit numbers one routine could have 60, another could have 10 causing 0 score for both individual routines but the whole program would have 70 causing a score of $7 \times w_i$ for that criterion.

For all other criteria the score is simply the sum of the individual scores for each routine. For the total score for the whole program this is added to the score for the maximum unit number criterion.

Another consideration in the portability function is the number of lines of code in the source program. Here, a simple count of the number of lines of code is taken, before any tidying or removal of continuation lines. Since one of the criterion is free format source statements continuation lines are counted as separate lines. A program with continuation

lines will have a higher portability than one where the lines are greater than 72 characters. This point could be argued and the number of lines could be taken after the continuation lines have been removed however the following example explains why they have been considered separately. Take a Fortran statement which covers three lines i.e. one line with two continuation lines. If each of these three lines contain 90 characters then the score will be 3 for 'free format source code' criterion, since there are 3 lines with more than 72 characters. If the number of lines in this case is taken to be 3 (i.e. count continuation lines separately) a score of $3 \times count$ will be given for this statement (where *count* is that for the 'long lines' criterion) which works out as $1 \times count$ per line which is fairly reasonable. If the number of lines is taken to equal the number of statements (i.e. ignore continuation lines) the number of lines will be 1 and so there will be a score of $3 \times count$ for 1 line which will give an exaggerated portability score.

The final value to be taken into consideration is the Portability Factor which is a measure of the accepted level of portability required of the particular program. This factor can be decided in advance and is basically the minimum number of lines affected by portability criteria for the program to be considered portable at all.

For example, with a factor of 2 the program will be given 0% portability if there is a score of 1 for every 2 lines of code. i.e. for a 10 line program a score of 5 = 0% portability. For a factor of 5 the program will have 0% portability if there is a score of 1 for every 5 lines of code. i.e. for the 10 line program a score of 2 = 0% portability.

Another way to look at this factor is that for a factor of 2 then the criteria scores are allowed to amount to $\frac{1}{2}$ of the program before 0% portability is reached. For a factor of 5 then the criteria scores can amount to $\frac{1}{5}$ of the program before 0% portability is reached.

This factor can be decided before any work is carried out and it is very much a 'user-defined' value. It should be decided in advance what should be considered as 0% portability i.e. how bad the portability needs to be before a program (or routine) is considered too bad to port.

For the NOMIS example it was decided that a routine should be considered to have 0% portability if the criteria score amounted to more than $\frac{1}{3}$ of the code. This gives a portability factor of 3 which was considered reasonable.

The amount of time and the number of programmers available should be taken into account when deciding this factor. With NOMIS a full-time programmer was available

to port the code with a timescale of up to three years so it could be said that if more than $\frac{1}{3}$ of the program was non-portable then this would not be feasible. If this programmer had not been available then a factor of 5 or even higher could have been chosen, meaning that if more than $\frac{1}{5}$ of the code was non-portable, there would have been 0% portability.

The portability factor can be explained simply as the proportion of code allowed to be affected by portability criteria i.e. if one line in three is allowed to be affected for 0% portability then a factor of of 3 would be taken. In fact, the effect of each criterion is not necessarily counted as one single line. A serious criterion, although possibly only occurring in one line of code, will give a higher score than 1 (i.e. 1-10) and this is the score considered above. For example, if there is one line with a criterion which has a weighting factor of 5 then this will have the same score as a program with 5 lines, each having a criterion with weighting factor 1. If there are 10 lines in the program then for this program to be 0% portable there would need to be a factor of $\frac{5}{10} = \frac{1}{2}$ rather than $\frac{1}{10}$ if simply the number of lines is considered. Therefore a portability factor 3 can be taken as meaning accept the program will be accepted if less than $\frac{1}{3}$ of it contains portability criteria, in reality there will probably be much less than $\frac{1}{3}$ of the lines affected by individual criteria but the seriousness of these criteria add up to $\frac{1}{3}$ of the program.

4.4.2. Portability Function

Taking all of these factors into consideration a function can now be defined for the portability of a particular routine, or the program as a whole. This function can be considered as follows:

$$Portability\ Function = 100 \times (1 - \frac{Total\ Score \times Portability\ Factor}{N^o\ lines\ of\ code})$$

or:

$$PF = 100 \times (1 - \frac{F}{L}(\sum_{i=1}^{n} x_i w_i + \sum_{j=1}^{k} s_j m_j))$$

Where:

PF = Portability of program or routine.

F = Portability Factor (Proportion of code in error for 0% portability).

L = Number of lines of code.

x_i = Count for criterion i.

w_i = Weighting factor for criterion i.

s_j = Occurrence of system specific subroutine j.

m_j = Weighting factor for system specific subroutine j

This gives what can be considered as the percentage portability of a particular program or routine. The range of values for this formula is $-\infty$ to 100. The negative percentages should be seen as a warning that the particular routine contains so many problems that portability may be too difficult to consider and the routine should be re-designed.

4.5. Criteria Counts and Weighting Factors

4.5.1. Execution of 'DO' Loops

It is not possible to detect exactly whether a 'DO' loop has a potential portability problem until run-time since it will depend on how the expression within the DO loop evaluates. As in the example given in Section 3.2.1., if the expression evaluates to 0 then discrepancies may occur. In this case some compilers will execute the statements within the loop once whereas others will not execute them at all even though the expression evaluates to 0.

In order to obtain a count for all DO loops which could potentially affect portability it would be necessary to count all the DO loops where the expression may evaluate to 0 or a negative value. Obviously DO loops such as:

```
DO 100 I=1,10
```

do not need to be included in the count since there is no way this DO loop could be executed anything other than 10 times. In the case of:

```
DO 100 I=1,ISUM
```

the result of the expression depends on the value of ISUM and so could potentially be evaluated to 0 or less. This DO loop should be included in the count.

From this it can be deduced that any DO loop where the expression contains at least one variable has potential for this problem to occur. An approximation to the number of

DO loops having the potential to affect portability is therefore the number of DO loops containing at least one variable in their expression.

This must be considered only as an approximation since a detailed examination of the code may reveal that possible values of the variables concerned may be such that the expression would always evaluate to a non-zero positive value. However, a detailed examination would involve work in itself so in turn could be counted as affecting the portability so in this discussion the inclusion of these DO loops in the count is not considered excessive.

The actual counting of these DO loops containing variables can be performed by using an awk script on the code and incrementing the count each time a DO loop is encountered with an expression containing anything other than numbers.

Although a relatively serious criterion if it occurs the weighting factor for DO loops containing variables is set at 2 which is relatively low. This is to minimise the effects of extra loops being included in the count due to the approximation of the count. The detection of this criterion by the use of static analysis techniques is, in general, not possible in all cases.

4.5.2. Free Format Source Statements

These are very obvious when looking at the code and it would be most likely that if free format source statements occurred in the code then the whole program would be written with free format. The count is taken as the number of lines of source code written with a free format so in most cases this would equal the number of lines of source code.

The weighting factor for this criterion is set at 2 since for an individual line of source code it would not be too difficult to convert it to standard format. When looking at the program as a whole and considering the amount of work required to convert the whole program to standard format it may appear that this criterion should have a higher weighting factor but the weighting factor is an indication of the difficulty in changing one line which is not serious. The score for this criterion obtained by multiplying the count by the weighting factor will be more dependent on the number of lines of code and therefore a large program which would require a lot of work to convert from free format will indeed have a large score, proportional to the number of lines of source code.

4.5.3. Format of Variable Names

This criterion can be split into two for obtaining counts. Firstly, any variables declared with lengths greater than 6 characters will produce a warning message when compiled with the Fortran VS compiler. The number of such warning messages can be counted and taken as the count for the number of variables with long names.

The other count obtained from this criterion is that of the number of variable names containing a $ as part of their name. In this case we consider the count to be the number of variables declared with names containing a $ as part of the name. It could be argued that the count should be the number of occurrences of a variable name containing a $ anywhere in the code, however to solve the problem (as in the case of long variable names) it would be necessary to replace the variable name with a more standard substitute. This would normally be done using some form of global substitution so while every occurrence of the variable needs to be replaced it would involve the same amount of effort to replace 10 occurrences of the variable as it would to replace 100. Each variable therefore has one count. A count of variables containing a $ can be obtained from the index produced by FTNTIDY using an awk script which searches the variable name part of the index for the $ symbol.

Both these criteria have a moderate weighting factor of 3. Although fairly easy to replace variable names globally, each variable only gives a count of 1 so the weighting factor has been set slightly higher to compensate for the lower counts. Care also needs to be taken when globally replacing the names since the same name may occur in text strings or as part of another variable name so the replacement is not trivial and therefore the weighting factor should be high enough to have a significant effect on the portability function.

4.5.4. Variable and Function Length Specifications

Whilst fairly difficult to solve, in-valid variable and function length specifications are simple to detect using compiler warning messages. A message will be produced for each function or variable declared non-standardly. As for variable name formats, the count is taken as the number of variables or functions declared non-standardly (obtained from the warning messages) rather than the number of occurrences of this variable or function in the code. The score for this criterion should be higher since the problem may not be as easily solved. In some cases it may be possible to simply change the declaration

statement, for example LOGICAL*1 can be replaced by CHARACTER or REAL*8 can be replaced by replaced by DOUBLE PRECISION. In other cases such as INTEGER*2 and COMPLEX*16 there is no direct replacement so the use of the variable must be considered before deciding on a replacement and the replacement may involve a lot of work.

For the purposes of the weighting factor, these variable types have not been distinguished between so, in order to average out the difference in work between simple replacements and large code changes a weighting factor of 4 is considered appropriate.

4.5.5. Data Initialisation not in DATA Statements

For a count of initialisation statements not in DATA statements it suffices to count all initialisation statements and detect those which do not occur within DATA statements. Before doing this the code should be run through FTNTIDY, all comment lines and statement labels removed and all continuation lines converted into single long lines. Any pairs of / symbols will signify either an initialisation or a COMMON block. If the statement is a COMMON block then the first word of the statement would be 'COMMON' and if a valid data initialisation statement then the first word in the statement would be 'DATA'. These rows should be eliminated from the count but any other rows should be counted as the best approximation to in-valid data initialisation statements.

Once detected, these statements are not normally too difficult to rectify, the initialisation part of the statement can be moved to a separate DATA statement, however this would not be a completely trivial process so a weighting factor of 2 is set.

4.5.6. Order of Statements

As previously noted, it is beyond the scope of a simple portability function to detect all deviations from the standard ordering recommendations so the ordering criterion considered here is that DATA statements must come after type declaration statements. When compiled, any DATA statement occurring before or within the type declaration statements will produce a warning message. The number of such warning messages is taken as the count for this criterion.

The criterion is given a weighting factor of 1 since the problem is fairly easy to detect from the warning messages and the solution is also quite simple. The DATA statements violating the standard need to be moved to a more suitable position which should be possible with a simple editor.

4.5.7. Hollerith Strings

A count of the number of Hollerith Strings in the code is again obtainable from the compiler warning messages. In this case, however, although a count is easily obtainable and the solution fairly straight forward - all Hollerith Strings need to be replaced with text strings - this criterion is given a significant weighting factor of 3 since it may be difficult and time-consuming to obtain the exact location of the Hollerith Strings within the code. The best way of detecting a Hollerith string would be to search for the occurrence of an integer immediately followed by the character 'H'. It is obvious though that many surplus statements may be found by searching in this way and this could be very time-consuming with a large piece of code.

4.5.8. Alternative Subroutine Calls

To detect this criterion it suffices to count the number of subroutine calls containing the & character in place of the * character. An approximation to this can be obtained using awk by removing all comment lines, text strings and converting continuation lines into single long lines. The awk script can then count all lines which contain the word CALL followed, somewhere on the same line, by the & character. It should be noted that it is the number of CALL statements containing the & character that are counted, not the number total number of & characters in CALL statements. For example:

CALL SUB(A,B,&99)

will give the same count (1) as:

CALL SUB(A,B,&99,&100,&101,&102).

This criterion is given a low weighting factor of 1 since it is fairly easy to detect and the correction is a simple replacement of & by *.

4.5.9. Character Handling Criteria

These criteria, while considered separately in Chapter 3, may be considered together here. To detect the occurrence of character variables in both COMMON blocks and EQUIVALENCE statements which do not contain soley character variables it is necessary to count the warning messages produced, one for each character variable in violation. It is easy therefore to detect the presence of these criteria and to work out the effect on portability but much more difficult to correct the problem. Major changes to the structure of the code may be necessary, particularly in the case where character variables

are equivalenced to other variable types. This criterion is therefore given a weighting factor of 4.

For character variables in COMMON blocks it is not quite so serious, hence a slightly lower weighting factor of 3 for this criterion. Each COMMON block will need to be replaced at all of its occurrences but replacement COMMON blocks should be possible without the need for major changes to the body of the code.

4.5.10. System Specific Subroutine Calls

This criterion is considered in a different way to the others being, by far, the most important. Rather than obtain a count of the number of system specific subroutine calls and multiplying this by a general weighting factor, each system specific subroutine call is considered individually. The EXTREFS program provides a list of system specific subroutines called by the program being considered. Each system specific subroutine called by the program is given its own individual score depending on whether this particular subroutine is widely available on other systems, is easy to re-write, or will require complete re-writing of parts of the code. These scores can be considered equivalent to the weighting factors for other criteria but cover a wider range, from 1 being the lowest to 10 being the highest. Note that a score of 0 is not possible since, by definition, a system specific subroutine must have some effect on portability or it wouldn't be specific to that particular system.

A score is given only once for the occurrence of a system specific subroutine call in the piece of code being considered, irrespective of how many calls there are to that subroutine in the piece of code. It is assumed that if a solution is found to the portability problems created by the subroutine call, then that solution will be valid for all occurrences of the subroutine call in the code. The score given to the criterion of system specific subroutines for the program is the sum of the scores given to each individual system specific subroutine called by the program.

Fig 4.6b at the end of this chapter shows the scores for a sample of system specific subroutines on the MTS system.

4.5.11. Internally Defined Filenames

In order to detect internally defined filenames declared in a non-standard way it is necessary to count the number of calls to the system specific subroutine FTNTIDY where the string passed as the first parameter starts with the characters 'ASSIGN'. This count can be obtained using a simple awk script, after first using FTNTIDY to format the source code correctly, then removing comment lines and converting continuation lines into single long lines. Note that in this case, text strings are not removed since the characters ASSIGN are themselves part of a text string.

It is also worth noting that the occurrence to FTNCMD will also be included in the count for system specific subroutines as FTNCMD is a system specific subroutine and may be used to execute any MTS operating system command, not just in the assignment of internally defined filenames. It would be possible, within the portability function, to deduct the score for FTNCMD as a system specific subroutine each time it was counted as an internal filename assignment to eliminate these duplicate counts, however, in this discussion no scores are deducted since it can further be argued that all calls to FTNCMD should be included in the count of system specific subroutine calls and those specific calls performing a filename assignment should also be included as a count of internally defined filenames.

The weighting factor for internally defined filenames has been set at 2 since, while not trivial, it is fairly simple to detect the appropriate calls to FTNCMD and then replace them with more standard OPEN statements.

4.5.12. Character Codes

The problem of detecting the use of ASCII or EBCDIC character codes in the text is a serious one in source code where it occurs. It is very difficult to define a system for detecting these character codes so in this case a very rough approximation is taken from the code after removing all comment lines, text strings, statement labels and converting all continuation lines into single long lines. The code is then searched (using an awk script) for any lines containing an integer in the range 64 - 250. Obviously this count will give a much higher value than the number of character codes there actually are in the code, however it is an indication of the number of lines of code which could potentially contain character codes.

In reality, this method of obtaining the count should only be used in code which does not have a strong mathematical basis (especially in code where there is a common use of the integers in the range 64 - 250), since this will give a very high count. It is advisable only to consider this criterion if there is a strong suspicion that there may be character codes within the source code.

Although this is a very serious criterion and almost impossible to detect, the correction is simple, replace character codes with the actual characters wherever possible and use the intrinsic functions CHAR and ICHAR. The weighting factor has therefore been set at 2 which may be considered fairly low. This has been chosen for a number of reasons:

(i) Although hard to detect the correction is relatively straight forward.

(ii) Due to the difficulty in obtaining the count, it is usual to obtain a count that is a gross over-estimation so by keeping the weighting factor lower, the effect of the error on the whole portability of the source code will be minimised.

(iii) For this criterion the emphasis is more on the count than the actual weighting factor. In other words the difficulty is more in detecting the errors (or obtaining the count) than on the actual seriousness of the errors.

4.5.13. Double Quotes Symbol

This is a near trivial criterion to detect and correct. The count is simply the number of lines of code containing pairs of double quotes symbols. This is done using an awk script after the code has been through FTNTIDY and comment lines and statement labels have been removed and continuation lines converted into single long lines.

Since the criterion is so simple to correct with the replacement of double quotes by single quotes (this could be done using a global edit) the weighting factor is given its lowest possible value of 1.

4.5.14. Default Unit Numbers

A count for the number of references to default unit numbers can be obtained from the index produced by FTNTIDY. An awk script can search for the 'LOGICAL I/O UNIT DICTIONARY' then pick out the required default unit numbers and count the number of references to each one.

Since, in many cases this criterion is irrelevant as a vast majority of systems use units 5 and 6 for read and write default units and it is very easy to replace the defaults

with those for another system if necessary, the weighting factor for the occurrence of default unit numbers is set at 1.

4.5.15. Statement Labels and Unit Numbers Limits

A count for the number of statement labels in a piece of code and the number of units referenced by a piece of code can be obtained from the index produced by FTNTIDY. In the case of statement labels, the count is accurate, in the case of units referenced the count may be an over-approximation since it is simply a count of the total number of units referenced by the code and doesn't take into consideration how many of these are open concurrently. The portability is only affected if the number of units open concurrently goes above the maximum allowed. Since there is no simple way of finding these exact figures, the total number of units referenced is taken as the best approximation.

For this reason the weighting factor for the unit numbers criterion is kept at 4 to reduce the errors caused by over-estimation. The weighting factor for the statement labels criterion is a maximum 5. Both of these weighting factors are high because the criterion is only considered if the counts go above the maximum allowable limits and if this is the case then the problems with the code are serious and may require a significant amount of work and re-writing of code.

4.5.16. Hexadecimal Data, Q and Z Format Codes

These criteria are considered together. The count for all three can be obtained from the compiler warning messages. The weighting factor for all three is set to 4 which is quite high. This is because, although easy to detect the occurrence of such criteria from the warning messages, it is not a trivial process to locate these occurrences in the code (particularly in the case of format codes) and the problem is certainly not trivial to correct. Major changes may be necessary to the code in order to avoid these criteria.

4.6. Summary

This chapter has described the simple type of static analysis tools required to obtain counts of the number of occurrences of each criterion considered in a particular section of source code. It has also discussed the value of weighting factor given to each of the criteria and the reasons behind these particular weighting factors.

Taking the counts and weighting factors for each criterion, together with the number of lines of source code it has been possible to define a weighted portability function

for the measure of portability of a program. Chapter 6 will show how the weighted portability function has been applied in practice to a real example, that of the NOMIS program.

Fig 4.6a shows the weighting factors (and codenames) given to each of the criteria considered. Fig 4.6b shows the weighting factors given to particular system specific subroutines. This table should contain a weighting factor for every system specific subroutine available on the system under consideration, however, this would be impractical so we have considered just those MTS Specific subroutines called by the NOMIS program.

Criteria	Criteria Code	Weighting Factor
Execution of DO loops	doloops	2
Free Format Source Statements	freeform	2
Free Format Source Statements	longlines	2
Format of Variable Names	longvar	3
Format of Variable Names	dollarinvar	3
Variable Length Specification	varlen	4
Function Length Specification	funclen	4
Data Initialisation	datainit	2
Order of Statements	dataorder	1
Hollerith Strings	hollerith	3
Alternative Subroutine Calls	ampersand	1
Character in Equivalence	cinequiv	4
Character in COMMON Blocks	cincommon	3
Internally Defined Files	intdeffiles	2
ASCII/EBCDIC Character Codes	ascii	2
Double Quotes Symbol	quotes	1
Default Unit 5	unit5	1
Default Unit 6	unit6	1
Statement Labels Limit	statements	5
Unit Numbers Limit	units	4
Hexadecimal Data	hexdata	4
Z Format Code	zformat	4
Q Format Code	qformat	4

Fig 4.6a: Weighting Factors for Criteria.

Subroutine	Weighting Factor
ADROF	5
ATNTRP	6
CMD	8
CMDNOE	8
CNTRL	5
COST	9
CREPLY	3
ERROR	5
FREEFD	5
FREESPAC	5
FTNCMD	8
GETFD	5
GETSPACE	5
GUINFO	5
LOADF	5
LSFILE	5
PAR	4
RCALL	5
READ	9
SERCOM	2
SETPFX	4
STARTF	5
SYSTEM	8
TIME	4
TOUCH	5
UNLDF	5
WRITE	9

Fig 4.6a: Weighting Factors for MTS Specific Subroutines.

Chapter 5. Portability Assistant

5.1. Introduction

This chapter describes how a relational database can be used to hold information about the criteria affecting the portability of all the subroutines comprising a Fortran program and provide an assistant for porting this program. A database alone is just a repository for information and therefore cannot be considered an assistant to portability. It should be noted therefore that the use of the term 'database' in the context of providing an assistant to portability includes the data repository together with the associated applications and programs used in data manipulation. Before describing the structure of the database in detail some general comments about the use of relational databases (in particular Ingres) are given.

5.1.1. Relational Databases and Ingres

Date [DATE87] gives a definition of a relational database:

"A relational database management system (relational DBMS for short) is a system that allows both end-users and application programmers to store data in and retrieve data from, databases that are perceived as collections of relations or tables."

He then extends this to say:

"A relational database is a system in which:

(a) The data is perceived by the user as tables (and nothing but tables); and

(b) The operators at the users' disposal (e.g. for retrieval) are operators that generate new tables from old. For example there will be one operation to extract a subset of the rows of a given table, and another to extract a subset of the columns - and of course a row subset and a column subset of a table can both in turn be regarded as tables themselves."

A detailed discussion of relational databases will not be given here other than to say simply that the term relation is a mathematical term for a table and a relational database can be considered as a collection of tables with each table containing items of data related to each other in some way.

The idea of a relational database suites the portability assistant example particularly well. All the information about the Fortran source code and the criteria affecting portability can be grouped together into tables. Operators can be defined (in this case the operators are applications and programs) which extract rows and columns from these tables, perhaps performing quite complicated operations on the data extracted. The results of these operations are further sets of related data which can themselves be grouped together into tables. This fits exactly the definition of a relational database.

There are a number of important points to consider about relational databases before grouping the data into tables.

(i) Relational databases do not allow repeating groups. That means, for every row and column position in every table there is exactly one data value (even if it is null) and never a set of values. For example the two tables in Fig 5.1a could be considered equivalent but only table (I) would be allowable in a relational database.

(I)

File	Routine
acctbig	cost
slinks	write
rline	read
acctbig	write
slinks	read

(II)

File	Routine
acctbig	cost, write
slinks	write, read
rline	read

Fig 5.1a: Possible database table structures.

This will be demonstrated later in the structure of the example database, where tables will be of the form shown in the first table.

(ii) The entire information content of a relational database is represented as explicit data values. There are no "links" or "pointers" connecting tables to one another. If there are links between the data in different tables then they are defined by means of further tables. An over-simplification of the example database demonstrates this point. There is a table containing the scores for particular criteria affecting portability, a table containing the portability scores for particular files. These two tables are linked together by a table indicating which files contain which portability criteria. This demonstrates the point but it will be seen later that these tables, in reality are more complex than this.

(iii) It is not absolutely necessary, but generally true, that each table contains a "unique identifier", that is, a column or combinations of columns in a table where the value of any particular row in the tale is unique with respect the the values for this column or columns in every other row of the table. This point is true throughout the particular example discussed later. In most tables the unique identifier is a single column of the table, usually either the "file" column or "criteria" column. In some cases the unique identifier spans two columns such as in the mtsroutines table where neither the "files" column nor the "mtsroutine" are unique on their own but the rows are unique across the two columns. In no case in the whole database is there an occurrence of a row which is identical to another row in the same table.

5.1.2. The Ingres Relational Database

As can be seen from the previous section the portability example is particularly suited to a relational database approach since the data subdivides very easily into different tables which can easily be linked to each other. The database is basically used to store the data in such a form that it can easily be retrieved and manipulated to produce information on the structure and portability of the application considered and subsets of it. The details about the criteria affecting portability and their scores split into well-defined tables, as does the information about the structure of the application and which criteria factors affect which parts of the code. The resulting portability information produced using the portability function also splits into tables (or one main results table).

The relational database Ingres satisfies the requirements of the application considered and was readily available on the Unix machines the code for the NOMIS example was being ported to. Ingres lends itself well to this job with well-defined tables and the ability to retrieve information very simply using Structured Query Language (SQL) statements, to write Applications so that a user can obtain information and query the database as required with little experience and the ability to perform fairly complicated operations on data from a combination of tables to produce the results of applying the portability function by using Structured Query Language Statements embedded in other programming languages. These techniques are described in more detail in Sections 5.1.3 and 5.1.4.

All of this data manipulation could be done by hand using programming languages and file manipulations but, for a large application like the NOMIS example, Ingres allows for much greater versatility. Date has a whole chapter outlining the "Advantages of

Ingres", many of which are relevant to this example but he sums up the main advantage of Ingres in his introduction to the chapter:

"If the advantages of a relational system such as INGRES must be summed up in a single word, that word is *simplicity* where by "simplicity" we mean, primarily, simplicity for the user. Simplicity, in turn, translates into usability and productivity."

5.1.3. Structured Query Language (SQL)

There are many ways of interrogating and manipulating the data in an Ingres database. The use of QBF (Query By Forms) is the simplest way of viewing the tables and updating by hand the data in a single table. However, to perform more complicated queries and updates some form of programming language is required. These more complicated queries can be performed using Structured Query Language (or SQL) commands. These can be run interactively or put together in the form of pre-written scripts. SQL is most useful in retrieving and updating information which may come from a combination of tables and which may be subject to a particularly complicated search condition.

The other advantage of SQL statements is that they can be embedded within the code of other programming languages such as Fortran, Pascal and C. In the application considered here the main portability program is written in Fortran with embedded SQL statements.

5.1.4. Ingres Applications

Another facility of Ingres is the ability to write Applications. Ingres/Applications is known as an "application generator" which is a tool for the rapid development of installation-specific applications. These are often referred to as fourth generation tools (machine code, assembler and high-level languages being the first three generations). The interface to the application generator is therefore known as a "fourth generation language" or 4GL. The application generator gives the designer a very high level development interface including database access, screen input/output, screen data manipulation as well as the normal arithmetic and control flow facilities of normal high-level programming languages. The application is developed using "Visual Programming" which is an interactive dialogue with the system rather than just writing code.

An Ingres 4GL Application is presented to the user as a hierarchical arrangement of frames, where each frame consists of a form and associated menu. A form is a visual

"display-screen" corresponding to a normal paper form. An Ingres form is made up of "fields" which are used for data entry and display, and "trim" which is any other static information displayed on the screen. Ingres forms also include "table fields" which allow multiple rows and columns of connected data to be displayed together. An Ingres frame is a combination of a form and a menu which is a list of operations that may be executed using the form. The menu appears at the bottom of the screen below the form. The selection of a menu operation performs some database operation such as an update or query then either returns to the same frame or calls up another. The code behind these menu operations is written in 4GL.

5.2. Structure of the Portability Database

The database was set up with three basic components:

(i) Details about the particular set of programs and routines to be considered. This set of tables are variable and will contain information about the particular application used.

(ii) Details about the criteria affecting portability, together with their particular weightings factors. The values in these tables are chosen independently from the particular application so these tables may be constant for a number (or all) applications.

(iii) Temporary tables used in the production of the results and the final results table. These tables are again variable and are formed from the result of applying the 'Portability Function' to the values in the two previous tables.

The tables are grouped into separate categories Application Tables, Criteria Tables and Results tables. These are described below. (Examples of the tables are given in Appendix 3.)

5.2.1. Application Tables

These tables contain all the information required about the application being considered, including the counts of the occurrences of particular criteria in each routine in the application.

The tables in this category are as follows:

5.2.1.1. Files

This table stores the filenames of the programs and routines which make up the application to be considered. It contains the following columns:

file: filename of the file to be considered.

5.2.1.2. Master

This is the Master table containing all the relevant information which has been obtained about each file or routine making up the application. All possible information is included, whether general information or particular criteria information. If any particular column is not relevant for the particular application being considered then it can be left blank. For example, in the NOMIS application the decimal memory allocation column is left blank since it is not used in any further manipulations. This field could also be derived from the Hex memory allocation column if so required.

This table contains the following columns:

file: filename of the file to be considered.

lines: number of lines of original source code (before any formatting).

hex memory: size of object code in Hexadecimal.

decimal memory: size of object code in decimal.

subfunc: number of internal subroutines and functions called.

extrefs: number of unresolved external references.

mts: number of system specific or library routines called.

criteria: number of occurrences of criteria (one column for each).

This master table is linked to the detail tables by the file column.

5.2.1.3. Subroutines

This is a detail table linked to the master table through the file column. There is an entry for each internal subroutine called by each file in the application so a file will occur in this table as many times as it has internal subroutine calls. It should be noted that this table is usually for reference only since internal subroutine calls do not normally adversely affect portability.

This table contains the following columns:

file: filename of the file to be considered.

routine: name of internal subroutine called.

5.2.1.4. Extrefs

This, again, is a detail table linked to the master table through the files column. It contains entries for unresolved external references for a particular file. As in the subroutine table, the number of occurrences of a particular 'file' in this table indicates the number of external references called by that file.

This table contains the following columns:

file: filename of the file to be considered.

extref: name of unresolved external reference.

5.2.1.5. Mtsroutines

This is another detail table linked to the master table through the files column. It contains entries for machine specific subroutine calls. These are system specific Fortran Library routines.

This table contains the following columns:

file: filename of the file to be considered.

mtsroutine: name of system specific subroutine called.

5.2.2. Criteria Tables

These tables contain information about the criteria likely to affect the portability of the application and the weightings given to these criteria in relation to each other.

The tables in this category are as follows:

5.2.2.1. Criteria

This table contains details of the portability factor for each criterion the code will be tested against. This portability factor is the weighting factor used in the routines for working out the portability of the application. It should be noted that all possible criteria are included in this table and that if they are not considered to have any effect

on portability then their 'critcount' can be set to 0. For example, in the NOMIS example external references are not system specific and therefore have no effect on portability so have a count of 0. Machine specific subroutine calls are also set to zero as there is no overall score for the occurrence of a machine specific subroutine, each individual subroutine is considered separately and these values are stored in a separate table.

This table contains the following columns:

criterion: Name of criterion considered.

critcount: Weighting Factor for criterion considered.

It should be noted that the rows in this table correspond to the columns of the master table for ease of data manipulation when working out the portability of a routine or application.

5.2.2.2. Mtssubr

This table could be described as a detail table for one row of the criteria table. It contains the weighting factor for all possible system specific subroutines that may be called. In an ideal situation this table would contain a row for every possible machine specific subroutine on the machine, however this may be impractical, and in most cases it would suffice to contain all system specific routines likely to be called by the routines and applications being tested for portability. In this table none of the counts should be 0 since the definition of a system specific subroutine implies that it would not be available in the same form on another machine and therefore must have some effect on portability.

This table contains the following columns:

mtsroutine: Name of system specific subroutine.

mtscount: Weighting factor for this subroutine.

It should be noted that the names of the routines in 'mtsroutine' should correspond to routines found in the 'mtsroutine' column of the 'Mtsroutines' table.

5.2.3. Results Tables

These tables include any tables used for holding results derived from the data in the Application and Criteria tables. This category could include any tables used temporarily for holding information during calculations or permanent tables holding results for future reference. In this database there is only one table in this category.

The table in this category is as follows:

5.2.3.1. Scores

This is the main results table containing portability details for each subroutine comprising the application. The table contains the same information as the output from the 'portability-count' program and its use is to give more versatility to the results.

This table contains the following columns:

file: filename of the file to be considered.

score: portability score for this file.

lines: number of lines of source code in this file.

portability: percentage portability of this file.

Having a table of results rather than a flat file means that details of portability can be obtained in many different forms. For example the results can be sorted alphabetically, in order of portability, in order of number of lines of code. Details can also be obtained for a single file or a subset of files. For example, a list of all those files with less than 30% portability can be very easily obtained using Ingres SQL statements.

5.3. Database Applications

5.3.1. Introduction

An Ingres Application can be defined as a tool for the rapid development of installation specific applications. It comprises a hierarchy of interlinked frames which contain visual forms which can be used for querying and updating tables in the database. Each form has an associated menu attached so that operations can be performed on the database.

The applications written for this database are concerned only with "Query" frames rather than "Update" frames. That is, they allow data and information from the database to be retrieved and displayed but not updated.

Two useful applications have been written using the information stored in the Application tables. Both of these applications can be used for obtaining information about the interaction between files and their subroutine calls. A complete description of each application is given below.

5.3.2. Applications

5.3.2.1. Mtsroutines

This application allows you to take a system specific subroutine and find out which of the routines making up the application contain calls to this subroutine. On running the application the frame shown in Fig 5.3a is displayed on the screen. Entering the name of the system specific subroutine (or a wildcard to pick up more than one routine), then selecting 'Find' will fill in the files table with the names of all files containing calls to this subroutine. Selecting 'Next' will pull up the list of files for the next system specific subroutine satisfied by the original query condition, if there is one. If an exact subroutine name is given as the query condition then 'Next' will not find another subroutine to satisfy the condition and will then return to the original menu, ready to accept another query. If the original query condition contains a wildcard which is satisfied by a number of subroutines then once the required subroutine has been found 'End' can be selected rather than 'Next' to return immediately to the first query menu.

The following example illustrates the use of this application when porting code. Consider a large application containing a lot of system specific subroutine calls which is being ported to a new system. A lot of time and effort may be spent providing a substitute for a particular system specific subroutine. This new replacement subroutine may be called in the same way but let us say there is one difference in the parameters passed across from the calling routine. It is therefore necessary to make a simple change to each routine which calls this subroutine. Rather than searching by hand through the code to find out which routines have calls to this particular subroutine, the Mtsroutines application can be used to find a list of routines which require modification in a fraction of the time.

Another use of this application is to check quickly whether a particular system specific subroutine is called at all by the program before time and effort is spent finding or writing a substitute on the new system.

```
                    MTS Routines

    mtsroutine ______________________


                +------------------------+
                |Files                   |
                |========================|
                |                        |
                |                        |
                |                        |
                |                        |
                |                        |
                |                        |
                |                        |
                |                        |
                |                        |
                |                        |
                |                        |
                |                        |
                +------------------------+


 Find  Clear  End
```

Fig 5.3a: Mtsroutines Application Frame.

5.3.2.2. Routines

This application performs the reverse of the Mtsroutines application. On running the application the frame shown in Fig5.3b is displayed on the screen. In this case the query is on 'File' which is one of the routines making up the application. The given file (or a wildcard query) is entered and on selecting 'Find' the mtsroutine table is filled in with any system specific subroutines called by this file. There is also an extrefs table which is filled in with details of any unresolved external references called by this file. Again if the original query condition is satisfied by a number if files then using 'Next' will step through these files giving system specific subroutine calls and unresolved external references called by each file. 'End' can be used at any point as before to return to the original query menu.

This application is used to find out more information about a particular file or group of files. For example, using the results of the 'portability' program (If say this was the only information available about the file) it may be found that two files had the same

```
                         FILES Table

    File: ____________________________________________

EXTREFS TABLE(S):              MTSROUTINES TABLE(S)

    +---------------+              +---------------+
    |Extref         |              |Mtsroutine     |
    |===============|              |===============|
    |               |              |               |
    |               |              |               |
    |               |              |               |
    |               |              |               |
    |               |              |               |
    |               |              |               |
    |               |              |               |
    |               |              |               |
    |               |              |               |
    |               |              |               |
    |               |              |               |
    |               |              |               |
    +---------------+              +---------------+


  Find   Clear   End
```

Fig 5.3b: Routines Application Frame.

portability score, but before deciding how to go about porting these files more information was required. Using the Routines application it may be found that one file has a lot of subroutine calls which are system specific and this is the main reason for its particular portability score. The other file may be found to have no system specific subroutine calls so it would be necessary to look further to find the reasons for its particular portability score.

Another example of a use for this application is if, say, only part of an application, or a particular file were to be moved to the new system possibly for testing the feasibility of moving the whole system. In the NOMIS example, all the code for file handling was moved over onto Unix as a test system since this was likely to be the most serious portability problem. Had this part of the of the application not been feasible on Unix then another operating system would have been considered. Therefore it was better to test only this small subset of the application before porting the whole application. Using Routines it can be found out first what external reference and system specific subroutines

are called by this subset of files so that these can be made available on the new system. Not all system specific subroutines on the old system will be required on the new system for the application to run and Routines can give an indication of which are required for either the whole application, or a subset of it to run. Back with the NOMIS example it was necessary to provide replacements for the READ and WRITE system specific subroutines initially while getting the file handling subset of NOMIS to work.

5.3.3. Observations

As can be seen, these applications do not perform any tasks which cannot be done by hand but they do improve the speed and efficiency with which information can be obtained and can save time and effort by enabling information to be obtained in advance and so avoid porting pieces of code which are unnecessary. They may be used to assist in the porting of a large application such as the NOMIS example, when a significant number of people may be working on different parts of the code at the same time. They can be used, particularly in the planning stages so particular, possibly self-contained, sections of code can be split into subsets, possibly to be ported by different members of the team. They also help give some idea of the complexity of porting the code.

Another use of these applications (not even necessarily connected to system portability) would be if a large application was passed on from one person or group to another. If such a database of information about the code was available then these applications could be used to help work out the structure of the code and find any possible unresolved external references before getting to the point of compiling the code. This leads on to possible uses of a database of this type as part of documenting code but since this is a digression from portability it will not be considered here. All that will be said is that if the production of such a database where included in the process of documenting the code then, as well as being a useful documentation tool, it would be invaluable if the code was ever to be ported to a new system.

It should also be noted that the applications described above are just a sample of Ingres applications that could be written to aid the porting of code. Ingres applications can be written with a varying degree of complexity, ranging from the simple applications shown here to complicated multi-frame applications which could be used to provide a complete summary of portability criteria for the whole application.

5.4. Database Programs

5.4.1. Introduction

From the Introduction to this chapter it has been seen that it is possible to use Structured Query Language (SQL) commands embedded within the code of other programming languages such as Fortran, C and Pascal. This can be very useful when performing some form of query on a database where the data requires modification or needs to be input or output in some particular format. This section describes some programs which have been written in Fortran containing embedded SQL statements and are useful in working out the portability of applications.

5.4.2. Portability-Count Program

This program is used to produce the results of applying the Portability Function described in Section 4.4 to the data in the database. It produces a report giving the portability of the required routine or application.

The program can be run in one of two ways, either interactively from the operating system (in this case Unix), or by taking its input from a file.

When the program is run interactively the user is prompted to enter the name of the routine the portability information is required for. The results are then given and another routine prompted for. The user terminates this loop by issuing an "End of File". The results have also been accumulated for all the files entered and these results are now given for the application as a whole. In this case the application comprises those routines considered during this run.

In the NOMIS example there are over 200 routines so it would not be practical to run the program interactively and enter each routine name by hand. The list of files can therefore be put into a file and this filename given as an input re-direction to the portability-count program. The results will then be generated and output for each individual routine then the application as a whole.

When using portability-count the results are not only output in the form of a report to the screen (or file if the output is re-directed) but the information is also placed in the scores table of the database for future reference. This table (as defined in Section 5.2.3) is then available for further viewing and queries if results are required for individual

routines or a subset of the application. The format of the results is shown in Appendix 2 and contains the following information:

Filename: The name of the routine considered.

Number of Lines: The number of lines of source code in the routine, where the number of lines of source code is counted before any tidying or manipulation is carried out on the file.

Score: The score for the routine, as worked out in Section 4.4 from the counts for each criterion affecting the routine and the criterion's weighting factor.

Portability: The portability of the routine as given by applying the Portability Function to the information obtained about the routine.

At the bottom of the report is given information about the application as a whole. The number of unit numbers referenced by the application as a whole is given and the score resulting from this is given. This is included because the score for this criterion is not simply the sum of the scores for this criteria over the whole application but is dependent on the number of units referenced by the application as a whole (See Section 3.2). Also given is the number of lines of source code for the application as a while, its overall score and its overall portability, again worked out using the Portability Function.

All the data required by the program is obtained from the database using the embedded SQL statements within the Fortran code. In most cases for any routine the score is worked out by a simple multiplication of count for each criterion (taken from the Master table) by the weighting factor for that criterion (taken from the Criteria table). The results are then summed to give the score for that routine.

If the Master table and Criteria table are taken as matrices then this can be seen as matrix multiplication to give the scores and the application of the Portability Function to the resulting matrix to give the final portabilities. This is summarised as follows (where S_i are the scores corresponding to the subroutines i):

$$\begin{pmatrix} S_1 \\ S_2 \\ S_3 \end{pmatrix} = \begin{pmatrix} M_{11} & M_{12} & M_{13} \\ M_{21} & M_{22} & M_{23} \\ M_{31} & M_{32} & M_{33} \end{pmatrix} \times \begin{pmatrix} C_1 \\ C_2 \\ C_3 \end{pmatrix}$$

which gives:

$$Score \begin{pmatrix} S_1 \\ S_2 \\ S_3 \end{pmatrix} = \begin{pmatrix} (M_{11}C_1 + M_{12}C_2 + M_{13}C_3) \\ (M_{21}C_1 + M_{22}C_2 + M_{23}C_3) \\ (M_{31}C_1 + M_{32}C_2 + M_{33}C_3) \end{pmatrix}$$

Where M_{ij} are rows and columns of the Master table and C_i are rows of the criteria table. The portabilities of each subroutine are therefore given by (where P_i are the portabilities corresponding to the subroutines i):

$$Portability \begin{pmatrix} P_1 \\ P_2 \\ P_3 \end{pmatrix} = PF \begin{pmatrix} S_1 \\ S_2 \\ S_3 \end{pmatrix}$$

Where PF is the application of the Portability Function to the scores matrix.

This is only an analogy since, in reality, the resulting scores aren't as simple as performing matrix multiplication on the complete Master and Criteria tables. Some columns in the Master table can be omitted from the multiplication at this point since they are included for reference only, such as the memory allocation in Hexadecimal, or they are required for other parts of the calculation such as the number of lines of source code which is required by the Portability Function. The machine specific subroutines column is also an exception since at this point the Mtsroutines table containing details of which machine specific subroutines are called must be brought into the calculation and this could be considered as being multiplied by the Mtssubr table which contains the weighting factors for system specific subroutines. The scores for the number of units referenced and the number of statement labels are not simply multiplied by the weighting factor since they are only relevant if they exceed particular values.

The matrices analogy shown in the diagrams gives an idea of how Fortran and embedded SQL can work together to produce much more detailed results than just SQL queries alone.

5.4.3. Portability-Order Program

This program takes the information contained in the Scores table for the all the routines in the application and outputs the same information as the Portability-Count program but this time in order of increasing portability.

The main use of this program is in comparisons between routines. It also makes it very easy to see which routines require no modifications i.e. have 100% portability

and which routines have the lowest portability 0% or even negative values of portability which means they have excessive scores for at least one criterion.

5.5. Summary

This chapter has considered in detail how an Ingres relational database can be used as an assistant in the portability of software. It has considered the basic structure of one such database and shown how database applications can be written very quickly and easily which allow information about the portability of the program to be obtained and displayed in a user-friendly way. It has also considered the use of embedded Structured Query Language (SQL) to enable the weighted portability function to be applied to the data, thus providing reports on the portability of the program. An example of how such a database has been used in practice is given in Chapter 6.

Chapter 6. Application

6.1. Introduction

Having looked at ways of detecting the criteria affecting the portability of Fortran programs and the use of an Ingres Relational Database as a portability assistant, this chapter considers how this can be applied to a particular example. The example chosen is the NOMIS program. It looks in detail at the particular portability problem posed by this example and the particular problems encountered while porting the program. It also shows how an Ingres database and weighted portability function can be used to give measures of the portability of the program as a whole and its component subroutines.

6.2. Overview of NOMIS

NOMIS is an acronym for the National Online Manpower Information System. The system was set up in 1978 by the University of Durham Geography Department and is run under contract to the Employment Department Group. It allows access to up-to-date Government Statistics on employment, population, unemployment, migration and Job Centre vacancies. The data covers the United Kingdom using a wide range of standard geographic areas from standard regions right down to the smallest scale of wards and postcode sectors. As well as producing standard data tables NOMIS has a range of analytical facilities including Change analyses, locational quotients, shiftshare analyses and a worksheet/spreadsheet facility.

There are approximately 500 sites around the country who have online access to NOMIS. These can be divided into five main groups:

(i) Central Government.

(ii) National and Regional Government.

(iii) Local Government.

(iv) Private Sector Consultancy.

(v) Academic Researchers.

These users log into the Durham machine for online access using PC's or terminals and in most cases a modem which connects them to Durham using either the Joint Academic Network (JANET) or Global Network Service (GNS) Dialplus which is run by

British Telecom. (This was formerly PSS). Once connected, all processing is performed on the Durham machine with the data being either output to the screen in the form of NOMIS tables, sent to a variety of line and laser printers at Durham, the output then being posted back to the user or downloaded to the users own PC from which it can be processed or printed locally. There is also a facility for producing maps from the data.

NOMIS is the main source of official Government statistics concerning the United Kingdom Labour Market and the data is stored in over 70 datasets. These are divided into the current series that are constantly being updated and the historical series for data which is no longer collected. The amount of data currently held on NOMIS is over 26 GB.

Each dataset has associated with it a geographic building block. These are combined to form aggregate areas and data can be accessed directly for these areas. These geographic areas range from Regions and Counties down to Wards and Postcode Sectors.

6.3. The Problem

The NOMIS program evolved since 1979 years on the MTS (Michigan Terminal System) operating system running on an Amdahl 5860 machine at Durham University. The NOMIS system is written almost entirely in Fortran with a small number (about 8) of data compaction routines written in IBM 370 Assembler, however the Fortran code relies heavily on certain MTS file handling features and system specific subroutines. The Fortran code comprises 276 separate subroutines with a total of 37873 lines of source code. When compiled the NOMIS program is approximately 3 MBytes in size.

The data files were also stored on the MTS system. There are approximately 70 different types of dataset, broken down by date. These datasets are stored in a compacted form and amount to 5 GBytes of storage space. If uncompacted this figure would increase to over 26 GBytes. A detailed description of the data compaction techniques used by NOMIS is given by Blakemore and Nelson [BLAK89]. The original data compaction routines were written in IBM 370 Assembler on the MTS system.

The NOMIS system is not a static system, new data is continually being made available, often meaning additions and changes to the Fortran code.

The problem considered here is the porting of the whole NOMIS system from the MTS system to a new Unix system running on a Sun Microsystems Sparc Server 2. The

timetable for the porting of the system was 3 years and 3 people were involved in the work (While at the same time continuing normal routine work on the system). Another important aspect of the problem was the transfer of users, the production of an equivalent accounting system and the continuity of the system and training of users during the actual move. While these are all of major importance it is the porting of the Fortran Code that is considered here, how the criteria affecting portability could be detected, making use of an Ingres database as described in Chapter 5 and how the Portability Function defined in Chapter 4 can be applied to these criteria to provide estimates of the portability problems affecting both the individual Fortran subroutines and the System as a whole.

6.4. Particular Portability Problems

6.4.1. Introduction

Whilst there were many criteria affecting the portability of the NOMIS program there were a number of criteria which had a more serious effect than others. This section gives a detailed description of a number of examples of these particular problems together with the solutions undertaken.

6.4.2. Data File I/O

The NOMIS system is based around the access of data from what are often very large data files. To improve efficiency in running the program it was therefore necessary to use some form of direct access I/O facility. The Fortran standard direct access I/O was not suitable since it insists on fixed length records. In the NOMIS data files the record length can range from a few bytes to several thousand bytes in the same file so to define the record length as the maximum in the file would create prohibitively large files.

On MTS this problem was easy to solve by using two system specific subroutines READ and WRITE. These routines allow direct access of variable length records using special key addresses specific to the MTS system. Whilst totally non-portable, the original version of NOMIS made extensive use of these subroutines in its main data access subroutines, since efficiency and conservation of file space were considered more important than portability.

There were 8 NOMIS subroutines making calls to the MTS READ routine and 6 making calls to the MTS WRITE routine so a replacement was needed that would require as little change to these calling routines as possible.

It took approximately 1 man/year from the identification of the problem to the production of an equivalent set of subroutines on the target Sun 4 Unix system. Since the whole NOMIS program was based on the access of data records, it was necessary to evaluate how this could be done before making final decisions on what the target machine was to be.

Initially the use of a Unix facility, ndbm, was evaluated. This provides a database of key/contents pairs which would, in effect, mirror the MTS data files with the unique MTS line number key mapped onto the ndbm key and the actual data records stored as the ndbm contents. The use of this tool was eventually rejected due to the exceptionally large data files produced. Similarly to Standard Fortran direct access I/O each record was stored with a line length equivalent to that of the longest record in the file, therefore giving enormous 'holes' in the data files. For some of the datasets with long records there was an increase of 5 times the size of the original file.

The eventual solution to this problem was to write a specific tool for NOMIS which mirrored the MTS direct access of variable length records facility as closely as possible. This tool, called 'dblib' was written in C was and set up specifically so that no changes were needed to the Fortran subroutines calling it. This did give rise to a number of redundant parameters passed between the subroutines. Rather than the data being stored in one file as on the MTS system which had its own internal key or line numbers, dblib has two files for each NOMIS dataset. A directory file is of fixed format and contains an address key (the same as the MTS line number key) and the offset into the data file which is the byte offset where that particular data record begins. Once opened this directory file is stored in memory using the Unix memory mapping facility to improve efficiency of access. The data file comprises a header containing the line number key (for checking purposes) and the length of the record followed by the data record. These headers and records are stored one after each other as a long string of bytes. When a record is to be read the line number key is passed from the Fortran subroutine and the offset in the data file obtained from the memory mapped directory. The header is then read from the data file, starting from this offset, the line number key checked and the record length obtained. The appropriate number of bytes of the data record are then read and passed back to the Fortran calling routine as with the MTS READ facility.

The dblib tool also contains a replacement for the MTS WRITE subroutine which works in a similar way. New or replacement records are appended to the end of the

data file and the directory file is re-written to correspond to these new records. This was not considered a serious disadvantage since, in general, not much writing of data occurs within the day-to-day running of the NOMIS program. Once created the datasets tend to be fairly static. Separate facilities were also written for converting data transferred from the MTS system into the dblib format and for creating the dblib files from scratch for new data.

This is just an overview of the dblib tool. It is a complex replacement for what, on the MTS system, were straightforward calls to system specific subroutines. The important point however, is that by re-writing the subroutines by hand for the particular purpose required, the actual Fortran subroutines did not require major modification. There was a need to replace normal Fortran OPEN and Close statements with special calls to special dblib OPEN and CLOSE routines. The disadvantage of dblib is that while routines written in C can be portable it was decided that the efficiency of the dblib routines was improved so much by using system specific memory mapping, this was done at the expense of future portability.

6.4.3. ASCII/EBCDIC Character Codes

A large proportion of the NOMIS program makes use of the actual integer character codes which, in the case of the MTS system, were EBCDIC character codes. The main use of these character codes is in the NOMIS subroutines providing the command interpreter for the instructions given to the program by the user. These codes are entirely system specific and the target Unix system has the ASCII character set rather than the EBCDIC character set. It was therefore necessary to detect and translate all occurrences of EBCDIC character codes throughout the code.

As described in Section 4.3, it is the detection of these character codes that is the difficult process since, once found, it is a trivial exercise to replace the EBCDIC code with its equivalent ASCII code. The approach recommended in Section 4.3 is to search the code for all integers in the range covered by the character set (64 - 250) then manually check whether each is a genuine integer value or an integer character code requiring conversion. NOMIS makes considerable use of integers in the source code, particularly as statement label references in GOTO statements so a large number of extra integers will be picked up using this method.

Subroutine	Lines of Source Code	EBCDIC codes used	Integers detected
R6	345	48	67
R52	354	0	8

Fig 6.4a: Sample results of method to detect ASCII/EBCDIC codes.

Figure 6.4a shows how this method was applied to two NOMIS subroutines of equivalent size, one which makes extensive use of ASCII/EBCDIC character codes and one which has no ASCII/EBCDIC character codes at all.

In the subroutine R6 containing the ASCII/EBCDIC violations 19 extra integers are detected whereas in the subroutine R52 containing no ASCII/EBCDIC violations only 8 integers are picked up. These results can be improved by applying the MTS FTNTIDY tool to the subroutine and changing the range of the statement labels so that they are outside the range of possible ASCII/EBCDIC character codes. In this case only 6 extra integers are picked up in R6 and only 1 extra integer in R52.

One refinement, in order to reduce the search space for this criterion, is to look for the integer values in a particular context. The context relevant here is that of expressions containing integers. These can occur in assignment statements, conditional statements, etc., in fact, in any executable statement. Filtering out the declaration statements should remove a number of spurious values. In practice, when this was tried on a sample of the NOMIS code the results obtained were the same as using the previous method. Using the same NOMIS subroutines as in the previous example 6 integers were detected in R6 and 1 in R52.

This method was not pursued since it would entail writing more complex code analysis tools than were considered necessary for this work.

6.4.4. Character Variables in COMMON Blocks

The use of character variables in the same COMMON block as variables of a type other than character was used extensively in the NOMIS program. This was a language extension available with the Fortran 77 compiler on the MTS system and the occurrence of all such COMMON blocks needed replacing before porting to the target Unix system. The solution was to create separate COMMON blocks for these character variables but with 407 variables affected by this criterion this was not a trivial task. Also, these COMMON blocks had to be replaced in every subroutine in which they occurred.

6.4.5. Data Compaction

The compaction and portability of the data itself is not relevant to this thesis, however it is worth mentioning briefly since it was a major problem. Due to its size the NOMIS data is stored and accessed in a compacted form where, depending on the size of the data values, the records of each dataset are coverted from integers and packed into 12-bit, 16-bit, 24-bit or 32-bit words, with the large amount of zero data values converted to single negative integers before packing. A full discussion of the NOMIS compaction techniques is given by Blakemore and Nelson [BLAK89]. The interesting point here is that the packing and unpacking subroutines were written for the MTS system in IBM 370 assembler so they were obviously not portable.

The solution was to convert these routines to Fortran. This was initially done using the known extension to the Fortran 77 standard, the use of equivalencing character with integer variables. Whilst this was not generally portable, the extension was known to be available on the target Unix system. However, later it was decided to re-write these routines again without making use of this extension and this has improved their portability.

6.4.6. Overview

This Section has covered just a few of the problems faced while porting the NOMIS program. The transfer of such large amounts of data as used by the program was also time consuming both in machine time and 'human' time. Each dataset had to be transferred between the machines and converted into a format suitable for use by the dblib tool. Another problem was that some files (mainly those storing information about geographical indices) contained a combination of binary data and text headings. These files were transferred using binary file transfer to conserve the data which meant that the text part was not converted from EBCDIC to ASCII. It was therefore necessary to write a conversion routine for reading the text part of the records.

Many of the criteria affecting portability discussed in earlier chapters occurred in different parts of the source code. This was detected using the methods given and also, in many cases, by 'trial and error' while testing the code (once it reached the point of compiling without errors). The correction of most of these problems involved manual changes to the code, many of which resulted in problems and changes required elsewhere in the code.

6.5. Portability Measures

6.5.1. Introduction

This section contains the results and measures which can be obtained from the Ingres database containing details about the criteria affecting the NOMIS program. First the general portability scores are considered, then this information is used to derive values for the portability of the NOMIS program using various portability functions.

6.5.2. Criterion Counts and Scores

Before considering the actual values for the portability of NOMIS some consideration should be given to the values for the various factors which are taken into account. Fig 6.5a is a summary of the counts for each of the portability criteria found in the NOMIS program as a whole:

This table contains some general information in the first four rows, then includes each criterion in turn together with the count obtained for each criterion. These counts were obtained as described in chapter 4 and are in some cases the total number of occurrences of that criterion in the program, in others there is a count of 1 for each file the criterion occurs in. In the case of statements and units defined the count given is the total, although in actual fact, the only relevant part of the count is that part greater than the maximum allowed for the particular machine. In the case of the Sun 4 which is the target machine for NOMIS the relevant part of the count for statement labels is more than 401 statement labels in any one subroutine (the value of 3391 in this table is therefore not relevant) and for units referenced is more than 63 over the program as a whole.

6.5.3. Hatton et al.'s Formula

Firstly, Hatton et al.'s formula is used. This is the simplest formula and takes mobility to be:

$$mobility = 100 \times [1 - \frac{(time\ taken\ to\ port)}{(time\ taken\ to\ write)}]$$

The NOMIS program was written over a period of approximately 9 man/years and ported over a period of 2 man/years giving a value of mobility to be 77.8% portable. Under this formula, with Hatton et al.'s measures of portability this makes NOMIS not portable or transportable but not low enough to be considered essentially non-portable.

Criteria	Count
Total Lines of Code	37873
Internal Subroutines	308
External References	318
MTS routines called	75
cinequiv	23
cincommon	407
varlen	41
funclen	0
longvar	0
dataorder	0
hexdata	6
zformat	0
qformat	0
hollerith	0
ampersand	0
datainit	147
dollarinvar	0
statements	3391
units	122
unit5	27
unit6	14
intdeffiles	3
quotes	14
doloops	445
freeform	2
longlines	44
ascii	620

Fig 6.5a: Summary of counts of portability criteria affecting NOMIS.

6.5.4. Tanaka's Formula

The other formula discussed was that of Tanaka. This formula is defined as:

$$mobility = 100 \times [1 - \frac{(number\ of\ modified\ LOC)}{(number\ of\ total\ LOC)} \times \alpha]$$

where:

$$\alpha = \frac{workload\ for\ modifying\ 1\ LOC}{workload\ for\ developing\ 1\ LOC}$$

With $\alpha = 3$ as in Tanaka's formula for the STYLE program, NOMIS mobility can be calculated as:

$$mobility\ of\ NOMIS = 100 \times [1 - \frac{1927}{37873} \times 3]$$

which gives the mobility of NOMIS as 84.74%, thus putting NOMIS into the range of being considered "transportable".

This latter figure is an approximation to the formula only since we take the count of the number of occurrences of the criterion as the count of the number of modified lines of code, in reality many more lines of code were probably modified. Also this measure takes no account of the difficulty in solving some of the problems. The workload in modifying one line of code is not constant for the different criteria so the addition of weighting factors for the criteria give a more realistic measure of the portability of NOMIS.

6.5.5. Weighted Portability Function

The weighted function for portability defined in Section 4.4 was also used with the NOMIS data. This formula defines:

$$Portability = 100 \times (1 - \frac{Total\ Score \times Portability\ Factor}{N^o\ lines\ of\ code})$$

Where the Total Score is the sum over all the criteria affecting the program or subroutine of the weighted scores for each criteria and the Portability Factor is a constant indicating the level of portability acceptance which in the NOMIS case is set to 3.

The counts were then taken, together with the weighting factors for each criterion to produce a weighted score for each NOMIS subroutine and these scores were put into the Portability Function to obtain values for the portability of each individual subroutine of NOMIS and for the NOMIS program as a whole.

Applying the Portability Function to the NOMIS program as a whole gives:

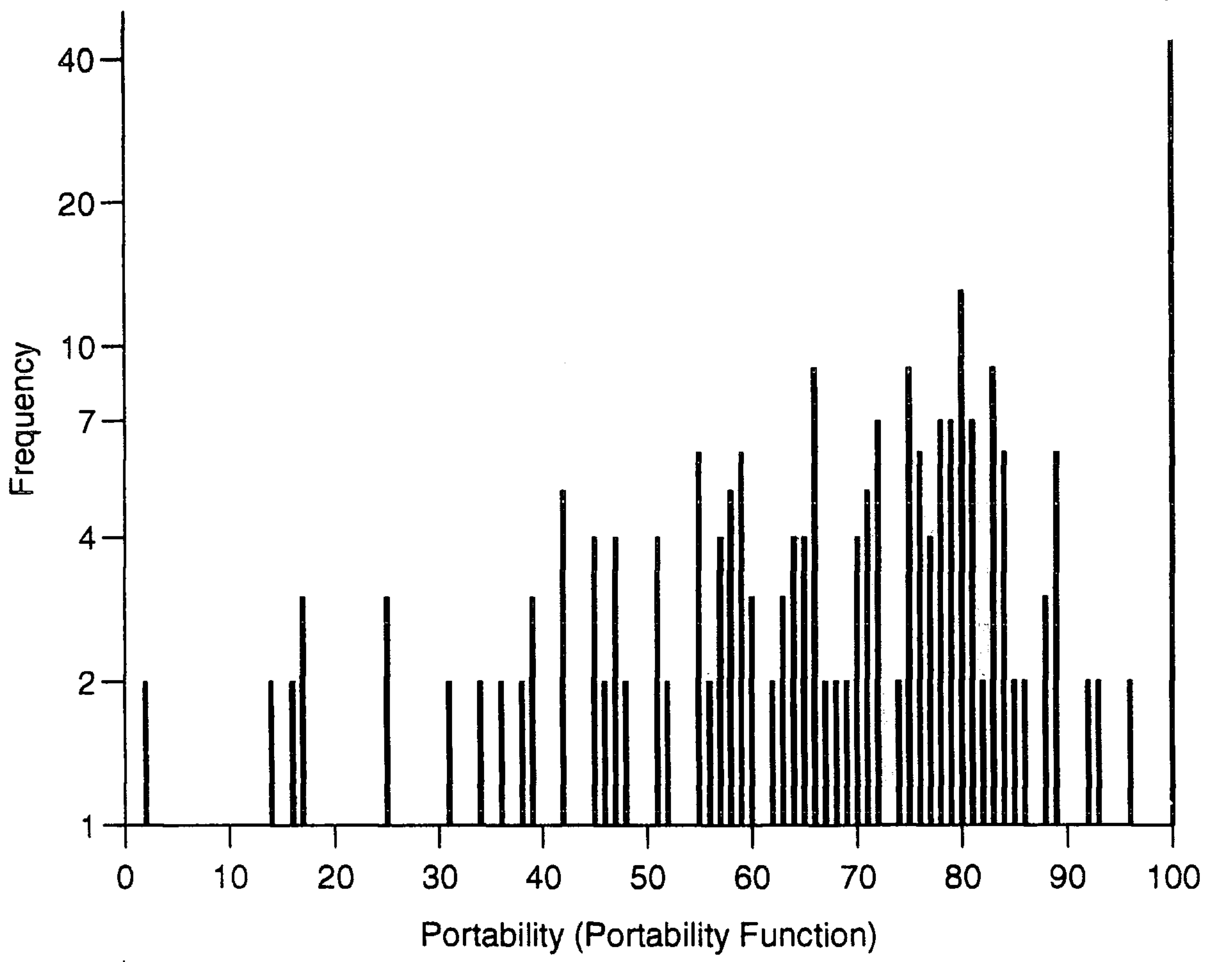
$$portability\ of\ NOMIS = 100 \times [1 - \frac{4823}{37873} \times 3]$$

This comes out as a portability of 61.8%. Using Hatton et al.'s measures of acceptable portability and taking this definition of portability to be equivalent to Hatton et al.'s definition of mobility NOMIS comes out as not portable or transportable but not bad enough to be considered essentially non-portable. By definition the use of a weighted function will give a lower portability than using a formula based on the number of modified lines of code so these levels of acceptance for portability can be relaxed more and any portability value greater than zero can be taken to indicate some level of portability, with any value over 50% indicating that the program is transportable. The NOMIS program was by no means highly portable due to serious problems with particular areas of code, however, a large amount of the code was ported without change so it could be considered transportable with 61.8% giving a fairly accurate measure of its portability.

6.5.6. Portability of Individual Subroutines

It is also interesting to consider portabilities of the individual subroutines which make up the NOMIS program. A breakdown of the portabilities of the individual subroutines is given in Appendix 2 together with the number of lines of source code, and the score obtained for the weighted sum of the portability criteria (without taking the number of lines of source code into account). The histograms in Fig 6.5b show the breakdown in frequencies of the portability values using both the Portability Function defined in this thesis and Tanaka's formula. Note that these histograms show portabilities with percentages greater than 0, although there are a number of routines where the portability problems are so bad that the routines are completely non-portable and produce negative portability values. These are considered in more detail later in this section.

The average portability for individual routines is 64.3% using the Portability Function and 85.4% using Tanaka's formula. In both cases this value is higher than that of the program as a whole, probably due to the added portability criterion of the number of unit numbers referenced which does not affect any of the individual routines but does affect (quite seriously) the portability of the program as a whole. The portability measures using the Portability Function are lower than those values using Tanaka's formula due to the weighting of the scores given for each criterion. These values are considered a more accurate indication of portability. Of the 276 subroutines making up the NOMIS



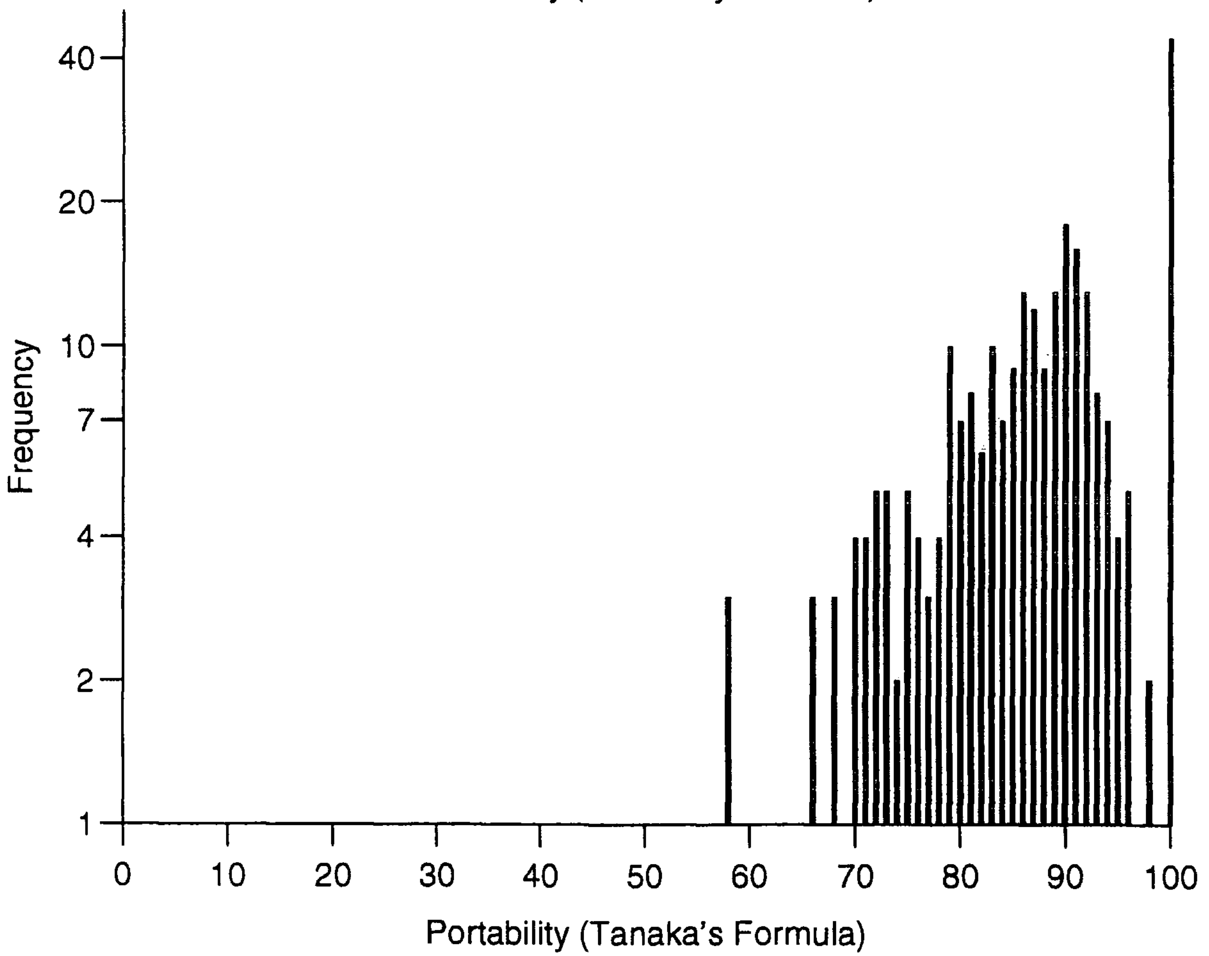


Fig 6.5b: Frequency of portability values for NOMIS subroutines.

program, 42 of them come out with 100% portability. This is 15.2% of subroutines which contained no criteria at all which affected portability and could be simply transferred from the MTS system and re-compiled without any change whatsoever. These are primarily small routines with the number of lines ranging from 0 to 101 apart from 1 routine with 226 lines.

The largest subroutine R92 has 1516 lines and this routine is also found to have the highest score for the effects of the criteria found, a score of 205. However it should be noted that this routine does not have a particularly low value for its portability, the value being 59%. This routine is used to produce results tables from the Census of Population, small area statistics data, the main component of its portability being possible ASCII/EBCDIC violations. In actual fact, although the occurrence of a large number of integers within the ascii/ebcdic range, 96 in fact, requires checking for ascii/ebcdic violations the integers in this routine are in fact used as straight integers and do not adversely affect the portability. However, as noted previously, the fact that these 96 integers have to be checked does impede the porting of the program and so the program cannot be considered 100% portable. Fig 6.5c shows a comparison between the portabilities of the five routines with the highest scores for their criteria and fig 6.5d the five routines with the largest number of lines, together with their portabilities. As can be seen R92 is at the top of both tables and R91 appears in both tables. In actual fact these two routines have the same portability. R2 which is the main NOMIS routine has the second largest score and also the second lowest portability.

Subroutine	Scores	Lines	Portability
R92	205	1516	59
R2	132	166	-138
R6	130	345	-13
R91	118	884	59
R24	96	377	23

Fig 6.5c: Subroutines with the highest criteria scores.

With such a large number of routines comprising NOMIS it is not possible here to consider the reasons behind the portability values for all the routines so we will look in detail at the five routines having the lowest portability scores. In actual fact these routines are some of the 10 routines which come out as having negative portability, due to the

Subroutine	Scores	Lines	Portability
R92	205	1516	59
R206	49	898	83
R91	118	884	59
R195	52	823	81
R157	45	581	76

Fig 6.5d: Subroutines with the highest number of lines of source code.

fact that the weighted function may consider one line of source code to be given a score which amounts to a portability problem equivalent to many lines of code. It is possible in the worst case to find that the portability of the routine is so bad that it amounts to a problem more serious then re-writing every line of code in that routine. In this case, the portability function may produce a negative value. This routine is effectively 0% portable but the negative values are kept in the discussion below so that the relative seriousness of the portability problems of these routines can be compared.

Routine R1, Portability -246%

This routine comes out with the lowest portability value and is therefore a highly non-portable routine. The purpose of the routine on the MTS system was to extract the full PAD address so that the NOMIS program could log where the call was coming from e.g. JANET or PSS. A summary of the criteria affecting this routine is given in Fig 6.5e.

Criteria affecting R1	Count
Lines of Code	52
MTS routines called	9
varlen	3
datainit	1
statements	2
doloops	1
ascii	1

Fig 6.5e: Summary of criteria affecting subroutine R1.

Whilst there are a number of criteria affecting the portability of this subroutine, the main criterion accounting for the extremely negative portability is that 9 system specific subroutines are called from a routine having only 52 lines of source code. This subroutine

is therefore highly system dependent and, since its purpose is to obtain information about the network, it is also highly dependent on the network. This subroutine was actually dropped from the Unix version of NOMIS because, while the information it provided was useful, the effort required to re-write it totally was too prohibitive and outweighed the usefulness of the information obtained.

Routine R2, Portability -138%

This is the main program that the users run when they execute NOMIS. Its function is to check the particular user has authority to run NOMIS and if so to initialise and load the system. A summary of the criteria affecting this routine is given in Fig 6.5f.

Criteria affecting R2	Count
Lines of Code	166
MTS routines called	5
cincommon	23
hexdata	1
statements	3
freeform	1
longlines	1
ascii	14

Fig 6.5f: Summary of criteria affecting subroutine R2.

The main criterion affecting the portability of this subroutine is the occurrence of 23 character variables in COMMON blocks containing variables of other types. The main initialisation for the whole NOMIS program is performed in this subroutine so the majority of COMMON blocks are initialised here. It was necessary to separate out the character variables and place them in newly created COMMON blocks which required a significant amount of effort. There were also a significant number (namely 5) system specific subroutines called from this subroutine, all of which needed replacing. Of these 3 were subroutines involved with loading whichever part of the system was required. There were also 14 integers which had to be checked for possible ascii/ebcdic character codes so despite this not being a particularly small subroutine (166 lines), the portability of the main NOMIS routine was very low.

Routine R3, Portability -125%

The purpose of this subroutine is to translate an upper case name into mixed case to improve readability. A summary of the criteria affecting this routine is given in Fig 6.5g.

Criteria affecting R3	Count
Lines of Code	93
ascii	35

Fig 6.5g: Summary of criteria affecting subroutine R3.

The portability criterion for this subroutine is fairly obvious. It is a relatively small subroutine (93 lines) but contains 35 integers which are possible ascii/ebcdic character codes. In reality, although some of these integers were statement labels in GOTO statements which were picked up as criteria because they fit into the limits of the ascii/ebcdic character codes, many of these integers were ascii/ebcdic character codes and were having a bad effect on the portability of the subroutine. These character codes were replaced by hand with their character equivalents making the subroutine more portable.

Routine R4, Portability -95%

This subroutine checks the "locking status" of the NOMIS logfile for a particular user identifier. As each user runs the NOMIS program a record of their commands is stored for future reference in their own version of a logfile. A user can only run one NOMIS session at any one time so the purpose of this subroutine is to use the status of the logfile to detect if a NOMIS session is already running and if so deny the user access to another session. A summary of the criteria affecting this routine is given in Fig 6.5h.

As can be seen from the table this subroutine is very small, having only 46 lines of source code. However it has a variety of criteria affecting its portability. While none of these criteria cause significant problems on their own, the combination of criteria in such a small subroutine gives it a much lower (in fact a negative value) portability compared with the effect the same criteria would have on a larger routine.

Routine R5, Portability -53%

This is a subroutine specific to a particular type of data and is used to read global statistical information from a file, performing initialisation s if required. A summary of the criteria affecting this routine is given in Fig 6.5i.

Criteria affecting R4	Count
Lines of Code	46
MTS routines called	2
hexdata	1
datainit	2
statements	3
units	1
unit6	2
quotes	1
ascii	3

Fig 6.5h: Summary of criteria affecting subroutine R4.

Criteria affecting R5	Count
Lines of Code	84
MTS routines called	3
cincommon	3
varlen	1
statements	4
units	1
ascii	2

Fig 6.5i: Summary of criteria affecting subroutine R5.

This routine again is fairly small at 84 lines and has a variety of criteria affecting it. The most notable is the occurrence of 3 system specific subroutines calls, READ/WRITE which enable direct access of variable length records which is an MTS feature not widely available and CMDNOE which allows operating system commands to be executed and is not so difficult to replace with the equivalent (although still system specific and not portable) subroutine on the Unix machine. There are also three character variables in COMMON blocks with other data types, quite a large number for such a small routine.

6.5.7. Overview

A number of measures which can be obtained from the information stored in the Ingres database about the criteria affecting the NOMIS program have been described. Three different formula were applied to the NOMIS criteria, two of them coming from papers considering the portability of similar programs and the third being the Portability

Function defined in this thesis from the experience gained in porting the NOMIS program. This formula gives NOMIS a portability of 61.8% which, taking into account the problems encountered in porting NOMIS, is considered an appropriate value. It can be concluded from this that the NOMIS program, while not highly portable, can be considered 'transportable'.

The portabilities of the individual subroutines have been compared with a detailed look at the five least portable subroutines. From this information it can be concluded that there are two factors having an equivalent effect on the portability of subroutines:

(i) The occurrence of just one criterion to such an extent that it alone can seriously effect the portability of the subroutine.

(ii) The occurrence of a large number of minor criteria which, on their own, would have little effect on portability but their combined effect is enough to render the subroutine highly non-portable.

Both of these factors become more apparent in the smaller subroutines where the effect of the criteria is increased and in many cases the changes necessary involve an almost complete re-write of the subroutine. These routines account for the negative portability values that occur when the effect of the criteria is more serious than re-writing every line of the subroutine.

6.6. Summary

This chapter has covered in detail the portability of the NOMIS program and considered how both an Ingres database can be used to assist with the porting process and how the weighted portability function can be used to obtain a measure of the portability of both the whole program and its component subroutines. These measures have also been compared with some measures taken using different formulae, in particular those defined in Section 2.4.

In conclusion, NOMIS as a whole program is given a portability of 61.8 % using the weighted portability function. This is considered an accurate measure since the NOMIS program can, in no way be described as highly portable, and yet should not be considered non-portable since many of its subroutines are highly portable. In fact 15.2% of NOMIS subroutines contain no criteria affecting their portability at all and are 100% portable. Of the 23 criteria considered, 15 of them were found to affect the NOMIS program in some

way. In some cases the criteria had a serious affect on the subroutines they occurred in, even requiring complete re-writes of a few subroutines, in other cases the criteria had a minor effect but often occurred in a large number of subroutines.

Chapter 7. Conclusions and Outlook

7.1. Introduction

The purpose of this chapter is to draw together the important points and conclusions obtained throughout this thesis and consider what further work may be possible. It considers firstly the criteria and measurements which may be obtained about the portability of programs, not necessarily restricting this to Fortran programs. It then considers the portability assistant described, its possible future use, and how it may be extended and modified to suit a number of related situations.

7.2. Portability Measures and Criteria

After identifying from the literature review (Chapter 2) a large number of criteria affecting portability, this thesis has considered 23 criteria which affect, in particular, the portability of programs written using the high level programming language Fortran. These criteria were identified as being those most relevant in the portability of Fortran programs written for the MTS system, however this discussion could be extended to cover those criteria affecting the portability of Fortran in general. This discussion is also restricted to the portability of one programming language, namely Fortran. Other programming languages, for example Pascal or C provide criteria affecting the portability of their programs. Fortran is considered, by some, to be a particularly well-defined language as far as portability is considered, however, as shown, there are certain statements which are not well defined by the Fortran standard and many system specific extensions are available. Other programming language standards also have omissions and extensions. In some cases these may give rise to the same criteria as Fortran, in other cases different. The methods and ideas covered in this thesis have been based on experiences with Fortran but could equally well be applied to other programming languages.

The criteria considered in this thesis have been those concerned with programming languages, however, there are other criteria affecting portability which are independent of the programming language used. Some of these more general portability criteria are included below for reference:

Machine Criteria

(i) Character handling - collating sequences, 8/6 bit characters

(ii) Arithmetic range and accuracy - integer/real number range reading and writing files

(iii) Accidental machine dependencies

(iv) Peripheral devices (printers etc)

(v) Performance variations

(vi) Word alignment

(vii) Word length

(viii) Arithmetic operations

(ix) Rounding (or truncation)

(x) Maximum size of integers

(xi) Base of real number systems

(xii) Precision

(xiii) Machine constants

(xiv) Sign extension

Operating System Criteria

(i) Run-time environment

(ii) Overlays

(iii) File system

(iv) Data structures (databases)

(v) Paging systems

(vi) Interrupt handling

(vii) I/O

(viii) Memory management

(ix) Storage layout

(x) Multi-user problems

(xi) Locking of files

(xii) Security

(xiii) Interpreter

(xiv) Memory allocation (eg NDBM)

(xv) Logical units

(xvi) Loader initialisation of memory

(xvii) Software tools (verifiers, filters and preprocessors)

Chapter 4 has considered how the portability of Fortran programs can be measured. A weighting factor has been given to each of the criteria considered. These weighting factors have been determined by considering the effort involved in detecting the occurrence of the criteria and the effort involved in adapting the code to work on the target system. The criteria of system specific subroutines requires a more detailed measure than a simple count since each system specific subroutine called becomes a criteria in its own right. Each individual system specific subroutine is given its own separate weighting factor. It is sufficient in this case to restrict these measures to those system specific subroutines affecting the NOMIS program described in Chapter 6. Theoretically, a weighting factor should be given to every system specific subroutine and library function available on the machine.

A number of formulae were given in Chapter 2 as methods of defining a measure of the percentage portability (or mobility) of programs. The formulae defined by Hatton et al., and Tanaka were applied to the NOMIS example and gave portabilities of 77.8% and 84.74% respectively. Hatton et al.'s formula is a very rough approximation based on time only and does not consider the changes to the code at all. Tanaka's formula is an extension of this, considering lines of code rather than time. This formula however does not take into account the effort involved in modifying each line of code. This effort depends on the particular criteria affecting that line of code. For this reason the weighted portability function is defined as a more accurate formula for determining portability. When this formula is applied to the NOMIS example it gives a value of 61.8% as the portability to the NOMIS program.

This weighted portability function contains a portability factor which is important in governing whether a routine will come out with negative portability. The higher the portability factor the greater the probability of obtaining negative portabilities. This can be explained simply in that if there were fewer programmers, time etc. the program would

have a high portability factor and be more likely to come out with a negative portability meaning it would be difficult to port within the constraints and would possibly not be feasible. If more programmers were brought in or the time limit increased then the portability factor could be lowered and the program may then have a positive portability. The program will still have the same Total Score and the same number of lines but will have increased portability, explaining why portability cannot simply be dependent soley on problems in the source code. These are important but other factors must also be taken into account. In one situation a program may be considered totally infeasible to port but in another where time is unlimited it may be simple (if time consuming) to port the same program. In this function these time and labour factors are lumped together in the 'Portability Factor' but the function could be developed further to consider these 'external' criteria separately.

It should be noted that in this case 'Internal' criteria can be taken as those concerned with portability problems within the source code. 'External' criteria can be taken as those more general factors affecting portability such as time, money, the number of available programmers and their experience. External criteria are independent of the programming language and machines whereas internal criteria are totally machine and language specific.

Chapter 6 also considers the portabilities of the component subroutines of NOMIS and looks in detail at some of the reasons behind their particular values. The conclusions drawn from these results are that there is not one single reason for a routine to have a very low (or even negative) portability. There are however two situations which have a particularly detrimental effect on portability:

(i) The repeated occurrence of one criterion which can seriously affect portability. For example, a large number of ASCII/EBCDIC character codes in one subroutine.

(ii) The occurrence of a large number of minor criteria which on their own would have little effect but their combination adds up to serious portability problems.

These factors are emphasised more in those routines having a smaller number of lines of source code.

7.3. Portability Assistant

This thesis has considered the use of an Ingres relational database as an assistant to the porting of Fortran programs. The structure described in Chapter 5 is that of the particular database used to hold the portability information about the subroutines comprising the NOMIS program. The tables described are specific to this example but could be modified or extended to hold information about the portability of other programs. The database tables are structured in such a way that changes to the criteria counted for, or the weighting factors given to each criterion can be modified to suit the required example.

It has been emphasised throughout this thesis that the provision of full Fortran analysis tools has not been the aim. Existing tools have been described and used to obtain the measures of criteria affecting the code. In the absence of availability of a suitable tool, manual methods have been used. An extension of this thesis would be to write full analysis tools for Fortran and provide the automatic generation of the application tables in the database. (Those tables containing the measures of criteria obtained). It would then be possible to apply the portability assistant and function to any Fortran program and obtain measures of this program's portability with the minimum of manual intervention.

It should also be possible to extend the database to include information about the location of criteria in the code so it could be used in the actual porting process to determine the exact location of the code which needs replacing. For some criteria it may be possible to write further tools which would enable the automatic replacement of offending code. This would only be possible for criteria which could be detected without error so would involve highly specified detection techniques. (Most of the techniques described in this thesis were for evaluation purposes only and contain some margin for error.)

The use of the portability assistant described as it was in this thesis is intended as an aid to portability rather than the solution of portability problems. In providing reports on the criteria affecting the portability of particular subroutines and a percentage portability for each, the subroutines can be ordered according to their portability or criteria. It can be seen immediately which subroutines have 100% portability and can therefore be ported without change, and also which subroutines are most likely to cause major problems. Using such a portability assistant on a large program, such as NOMIS, where a number of programmers may be working concurrently on the porting process, it should be possible to evaluate (approximately), in advance, the time and effort required

to port each component subroutine. This would be an aid to planning the porting process efficiently.

The use of the portability assistant has been described here in the porting of existing programs which have not necessarily been written with portability in mind. It would also have a use in the design of new programs, particularly when designing with portability in mind. It has been shown from the examples in Chapter 2 that even designing for portability cannot always produce completely portable programs. Most large programs will contain some form of system dependency as standard language definitions are not fully defined and the improvement in efficiency by some system specific features may be considered more important than complete portability. The use of modular programming methods is recommended, as is the documenting of portability violations. The database structure described here could be adapted to the design of programs and used to hold information on the areas of the program which could be considered non-portable. This information would then be readily available when the program was ported. Had this information been available at the time the NOMIS program was ported, the porting process would have been performed more efficiently. The use of a database in such a situation would been that information on the structure of the program was available at all times. This would be an advantage at times other than when porting the program.

Appendix 1. Sample Results from FTNTIDY

```
OVERRIDING PAR=NOSOURCE

FTNTIDY OPTIONS: NOSOURCE FORMAT=EDITED NOBCD ISN XREF NOLBLXREF LINECNT=60
        ERRMAX=25 SPACE RELABEL NOFMTMOVE NOSEQ INCR=10 START=10 NODOCOMMENT
        INDENT=2 RTMARG=72 LBLJUST=RIGHT HOLQUOTE CONTCHAR=' ' DECK LIST
SCARDS=msc.testfile
SPRINT=msc.index
SPUNCH=msc.tidyfile
      MTS            INTERNAL        **** F T N T I D Y ****
    LINE NO.         STMT NO.        PUNCH LISTING
       1.               1            PROGRAM TEST1
       2.               2            INTEGER $COUNT
       3.               3            REAL COUNT, TEST, DO$DO
       4.               4            CHARACTER*20 FILE /'MSCFILE for testing'/
       5.               5            DATA TEST /2.77/
       6.                    C
       7.                    C       start of main program
       8.                    C
       9.               6            OPEN (UNIT=1,FILE=FILE)
      10.               7            CALL GET(UNIT, DO$DO, 3, 6, &10, &20)
      11.               8         10 CALL VALUE(COUNT, TEST, &30)
      12.               9            IF ($COUNT .EQ. 0) GO TO 10
      13.                    C       AFTER END OF LOOP
      14.              10            DO 20 , I = 1, $COUNT
      15.              11              PRINT *, 'HELP'
      16.              12         20 CONTINUE
      17.                    C       $continue with now commented out
      18.                    C       $CONTINUE WITH MTSPROG
      19.              13         30 STOP
      20.              14            END
            ***   SUBPROGRAM  DICTIONARY   ***
  NAME     TYPE  ATTR       REFERENCES

  GET            SUBR        7
  TEST1          PGM         1D
  VALUE          SUBR        8
  <EXIT>         SUBR       13
              ***   VARIABLE  DICTIONARY   ***
  NAME     TYPE  ATTR  COMMON  REFERENCES

  $COUNT   I*4                        2D       9       10
  COUNT    R*4                        3D       8?
  DO$DO    R*4                        3D       7?
  FILE     CHAR                       4D       6
  I       (I*4)                      10*
  TEST     R*4                        3D       5        8?
  UNIT    (R*4)                       7?
            ***  STATEMENT LABEL DICTIONARY   ***
   LABEL    DEFN       TYPE   ORIG     REFERENCES

     10       8               99       7       9
     20      12              100       7      10
```

```
    30      13              999       8
            ***  LOGICAL I/O UNIT DICTIONARY   ***
  UNIT                    REFERENCES

     1          6
TYPES: I=INTEGER, R=REAL, L=LOGICAL, C=COMPLEX, GEN.=GENERIC, N.L.=NAMELIST,
       FMT=FORMAT
       TYPES ENCLOSED WITHIN PARENTHESES INDICATE IMPLICIT DECLARATION
ATTRIBUTES: SUBR=SUBROUTINE, FCN=FUNCTION, S.F.=STATEMENT FUNCTION
REFERENCES: *=VALUE CHANGED, ?=SUBPROGRAM ARGUMENT, D=DEFINED, E=EQUIVALENCE,
            C=COMMON, R=READ, W=WRITE, M=MOTION
```

Appendix 2. Portability of NOMIS Subroutines

R1	Lines:	52	Scores:	60	Portability (%):	-246
R2	Lines:	166	Scores:	132	Portability (%):	-138
R3	Lines:	93	Scores:	70	Portability (%):	-125
R4	Lines:	46	Scores:	30	Portability (%):	-95
R5	Lines:	84	Scores:	43	Portability (%):	-53
R6	Lines:	345	Scores:	130	Portability (%):	-13
R7	Lines:	110	Scores:	39	Portability (%):	-6
R8	Lines:	115	Scores:	41	Portability (%):	-6
R9	Lines:	147	Scores:	51	Portability (%):	-4
R10	Lines:	81	Scores:	28	Portability (%):	-3
R11	Lines:	152	Scores:	50	Portability (%):	1
R12	Lines:	108	Scores:	35	Portability (%):	2
R13	Lines:	154	Scores:	50	Portability (%):	2
R14	Lines:	44	Scores:	14	Portability (%):	4
R15	Lines:	54	Scores:	16	Portability (%):	11
R16	Lines:	63	Scores:	18	Portability (%):	14
R17	Lines:	77	Scores:	22	Portability (%):	14
R18	Lines:	50	Scores:	14	Portability (%):	16
R19	Lines:	155	Scores:	43	Portability (%):	16
R20	Lines:	58	Scores:	16	Portability (%):	17
R21	Lines:	87	Scores:	24	Portability (%):	17
R22	Lines:	124	Scores:	34	Portability (%):	17
R23	Lines:	99	Scores:	27	Portability (%):	18
R24	Lines:	377	Scores:	96	Portability (%):	23
R25	Lines:	24	Scores:	6	Portability (%):	25
R26	Lines:	85	Scores:	21	Portability (%):	25
R27	Lines:	162	Scores:	40	Portability (%):	25
R28	Lines:	98	Scores:	24	Portability (%):	26
R29	Lines:	62	Scores:	15	Portability (%):	27
R30	Lines:	60	Scores:	14	Portability (%):	30
R31	Lines:	74	Scores:	17	Portability (%):	31
R32	Lines:	140	Scores:	32	Portability (%):	31

R33	Lines:	163	Scores:	36	Portability (%):	33
R34	Lines:	106	Scores:	23	Portability (%):	34
R35	Lines:	133	Scores:	29	Portability (%):	34
R36	Lines:	99	Scores:	21	Portability (%):	36
R37	Lines:	199	Scores:	42	Portability (%):	36
R38	Lines:	44	Scores:	9	Portability (%):	38
R39	Lines:	254	Scores:	52	Portability (%):	38
R40	Lines:	50	Scores:	10	Portability (%):	39
R41	Lines:	50	Scores:	10	Portability (%):	39
R42	Lines:	65	Scores:	13	Portability (%):	39
R43	Lines:	52	Scores:	10	Portability (%):	42
R44	Lines:	52	Scores:	10	Portability (%):	42
R45	Lines:	84	Scores:	16	Portability (%):	42
R46	Lines:	193	Scores:	37	Portability (%):	42
R47	Lines:	341	Scores:	65	Portability (%):	42
R48	Lines:	135	Scores:	25	Portability (%):	44
R49	Lines:	44	Scores:	8	Portability (%):	45
R50	Lines:	160	Scores:	29	Portability (%):	45
R51	Lines:	221	Scores:	40	Portability (%):	45
R52	Lines:	354	Scores:	64	Portability (%):	45
R53	Lines:	45	Scores:	8	Portability (%):	46
R54	Lines:	181	Scores:	32	Portability (%):	46
R55	Lines:	63	Scores:	11	Portability (%):	47
R56	Lines:	68	Scores:	12	Portability (%):	47
R57	Lines:	98	Scores:	17	Portability (%):	47
R58	Lines:	153	Scores:	27	Portability (%):	47
R59	Lines:	76	Scores:	13	Portability (%):	48
R60	Lines:	146	Scores:	25	Portability (%):	48
R61	Lines:	106	Scores:	18	Portability (%):	49
R62	Lines:	198	Scores:	33	Portability (%):	50
R63	Lines:	37	Scores:	6	Portability (%):	51
R64	Lines:	37	Scores:	6	Portability (%):	51
R65	Lines:	81	Scores:	13	Portability (%):	51
R66	Lines:	187	Scores:	30	Portability (%):	51
R67	Lines:	51	Scores:	8	Portability (%):	52

R68	Lines:	348	Scores:	55	Portability (%):	52
R69	Lines:	126	Scores:	19	Portability (%):	54
R70	Lines:	68	Scores:	10	Portability (%):	55
R71	Lines:	88	Scores:	13	Portability (%):	55
R72	Lines:	114	Scores:	17	Portability (%):	55
R73	Lines:	122	Scores:	18	Portability (%):	55
R74	Lines:	174	Scores:	26	Portability (%):	55
R75	Lines:	238	Scores:	35	Portability (%):	55
R76	Lines:	152	Scores:	22	Portability (%):	56
R77	Lines:	351	Scores:	51	Portability (%):	56
R78	Lines:	28	Scores:	4	Portability (%):	57
R79	Lines:	57	Scores:	8	Portability (%):	57
R80	Lines:	196	Scores:	28	Portability (%):	57
R81	Lines:	205	Scores:	29	Portability (%):	57
R82	Lines:	29	Scores:	4	Portability (%):	58
R83	Lines:	43	Scores:	6	Portability (%):	58
R84	Lines:	100	Scores:	14	Portability (%):	58
R85	Lines:	328	Scores:	45	Portability (%):	58
R86	Lines:	576	Scores:	79	Portability (%):	58
R87	Lines:	103	Scores:	14	Portability (%):	59
R88	Lines:	171	Scores:	23	Portability (%):	59
R89	Lines:	261	Scores:	35	Portability (%):	59
R90	Lines:	275	Scores:	37	Portability (%):	59
R91	Lines:	884	Scores:	118	Portability (%):	59
R92	Lines:	1516	Scores:	205	Portability (%):	59
R93	Lines:	30	Scores:	4	Portability (%):	60
R94	Lines:	45	Scores:	6	Portability (%):	60
R95	Lines:	229	Scores:	30	Portability (%):	60
R96	Lines:	157	Scores:	20	Portability (%):	61
R97	Lines:	32	Scores:	4	Portability (%):	62
R98	Lines:	56	Scores:	7	Portability (%):	62
R99	Lines:	33	Scores:	4	Portability (%):	63
R100	Lines:	57	Scores:	7	Portability (%):	63
R101	Lines:	73	Scores:	9	Portability (%):	63
R102	Lines:	68	Scores:	8	Portability (%):	64

R103	Lines:	144	Scores:	17	Portability (%):	64
R104	Lines:	193	Scores:	23	Portability (%):	64
R105	Lines:	227	Scores:	27	Portability (%):	64
R106	Lines:	35	Scores:	4	Portability (%):	65
R107	Lines:	86	Scores:	10	Portability (%):	65
R108	Lines:	87	Scores:	10	Portability (%):	65
R109	Lines:	167	Scores:	19	Portability (%):	65
R110	Lines:	36	Scores:	4	Portability (%):	66
R111	Lines:	36	Scores:	4	Portability (%):	66
R112	Lines:	36	Scores:	4	Portability (%):	66
R113	Lines:	90	Scores:	10	Portability (%):	66
R114	Lines:	133	Scores:	15	Portability (%):	66
R115	Lines:	162	Scores:	18	Portability (%):	66
R116	Lines:	230	Scores:	26	Portability (%):	66
R117	Lines:	319	Scores:	36	Portability (%):	66
R118	Lines:	429	Scores:	48	Portability (%):	66
R119	Lines:	84	Scores:	9	Portability (%):	67
R120	Lines:	177	Scores:	19	Portability (%):	67
R121	Lines:	96	Scores:	10	Portability (%):	68
R122	Lines:	317	Scores:	33	Portability (%):	68
R123	Lines:	99	Scores:	10	Portability (%):	69
R124	Lines:	205	Scores:	21	Portability (%):	69
R125	Lines:	62	Scores:	6	Portability (%):	70
R126	Lines:	81	Scores:	8	Portability (%):	70
R127	Lines:	210	Scores:	21	Portability (%):	70
R128	Lines:	232	Scores:	23	Portability (%):	70
R129	Lines:	158	Scores:	15	Portability (%):	71
R130	Lines:	171	Scores:	16	Portability (%):	71
R131	Lines:	310	Scores:	29	Portability (%):	71
R132	Lines:	312	Scores:	30	Portability (%):	71
R133	Lines:	387	Scores:	37	Portability (%):	71
R134	Lines:	43	Scores:	4	Portability (%):	72
R135	Lines:	88	Scores:	8	Portability (%):	72
R136	Lines:	97	Scores:	9	Portability (%):	72
R137	Lines:	110	Scores:	10	Portability (%):	72

R138	Lines:	118	Scores:	11	Portability (%):	72
R139	Lines:	132	Scores:	12	Portability (%):	72
R140	Lines:	531	Scores:	49	Portability (%):	72
R141	Lines:	94	Scores:	8	Portability (%):	74
R142	Lines:	178	Scores:	15	Portability (%):	74
R143	Lines:	48	Scores:	4	Portability (%):	75
R144	Lines:	49	Scores:	4	Portability (%):	75
R145	Lines:	85	Scores:	7	Portability (%):	75
R146	Lines:	99	Scores:	8	Portability (%):	75
R147	Lines:	99	Scores:	8	Portability (%):	75
R148	Lines:	110	Scores:	9	Portability (%):	75
R149	Lines:	110	Scores:	9	Portability (%):	75
R150	Lines:	120	Scores:	10	Portability (%):	75
R151	Lines:	231	Scores:	19	Portability (%):	75
R152	Lines:	50	Scores:	4	Portability (%):	76
R153	Lines:	76	Scores:	6	Portability (%):	76
R154	Lines:	194	Scores:	15	Portability (%):	76
R155	Lines:	260	Scores:	20	Portability (%):	76
R156	Lines:	316	Scores:	25	Portability (%):	76
R157	Lines:	581	Scores:	45	Portability (%):	76
R158	Lines:	27	Scores:	2	Portability (%):	77
R159	Lines:	66	Scores:	5	Portability (%):	77
R160	Lines:	92	Scores:	7	Portability (%):	77
R161	Lines:	335	Scores:	25	Portability (%):	77
R162	Lines:	28	Scores:	2	Portability (%):	78
R163	Lines:	28	Scores:	2	Portability (%):	78
R164	Lines:	28	Scores:	2	Portability (%):	78
R165	Lines:	56	Scores:	4	Portability (%):	78
R166	Lines:	137	Scores:	10	Portability (%):	78
R167	Lines:	171	Scores:	12	Portability (%):	78
R168	Lines:	263	Scores:	19	Portability (%):	78
R169	Lines:	88	Scores:	6	Portability (%):	79
R170	Lines:	89	Scores:	6	Portability (%):	79
R171	Lines:	100	Scores:	7	Portability (%):	79
R172	Lines:	101	Scores:	7	Portability (%):	79

R173	Lines:	222	Scores:	15	Portability (%):	79
R174	Lines:	264	Scores:	18	Portability (%):	79
R175	Lines:	287	Scores:	20	Portability (%):	79
R176	Lines:	31	Scores:	2	Portability (%):	80
R177	Lines:	60	Scores:	4	Portability (%):	80
R178	Lines:	61	Scores:	4	Portability (%):	80
R179	Lines:	75	Scores:	5	Portability (%):	80
R180	Lines:	75	Scores:	5	Portability (%):	80
R181	Lines:	105	Scores:	7	Portability (%):	80
R182	Lines:	108	Scores:	7	Portability (%):	80
R183	Lines:	109	Scores:	7	Portability (%):	80
R184	Lines:	109	Scores:	7	Portability (%):	80
R185	Lines:	110	Scores:	7	Portability (%):	80
R186	Lines:	110	Scores:	7	Portability (%):	80
R187	Lines:	154	Scores:	10	Portability (%):	80
R188	Lines:	313	Scores:	20	Portability (%):	80
R189	Lines:	32	Scores:	2	Portability (%):	81
R190	Lines:	33	Scores:	2	Portability (%):	81
R191	Lines:	79	Scores:	5	Portability (%):	81
R192	Lines:	116	Scores:	7	Portability (%):	81
R193	Lines:	148	Scores:	9	Portability (%):	81
R194	Lines:	161	Scores:	10	Portability (%):	81
R195	Lines:	823	Scores:	52	Portability (%):	81
R196	Lines:	69	Scores:	4	Portability (%):	82
R197	Lines:	242	Scores:	14	Portability (%):	82
R198	Lines:	89	Scores:	5	Portability (%):	83
R199	Lines:	91	Scores:	5	Portability (%):	83
R200	Lines:	92	Scores:	5	Portability (%):	83
R201	Lines:	93	Scores:	5	Portability (%):	83
R202	Lines:	107	Scores:	6	Portability (%):	83
R203	Lines:	181	Scores:	10	Portability (%):	83
R204	Lines:	243	Scores:	13	Portability (%):	83
R205	Lines:	358	Scores:	20	Portability (%):	83
R206	Lines:	898	Scores:	49	Portability (%):	83
R207	Lines:	38	Scores:	2	Portability (%):	84

R208 Lines: 59 Scores: 3 Portability (%): 84

R209 Lines: 79 Scores: 4 Portability (%): 84

R210 Lines: 79 Scores: 4 Portability (%): 84

R211 Lines: 96 Scores: 5 Portability (%): 84

R212 Lines: 415 Scores: 22 Portability (%): 84

R213 Lines: 84 Scores: 4 Portability (%): 85

R214 Lines: 120 Scores: 6 Portability (%): 85

R215 Lines: 43 Scores: 2 Portability (%): 86

R216 Lines: 152 Scores: 7 Portability (%): 86

R217 Lines: 71 Scores: 3 Portability (%): 87

R218 Lines: 79 Scores: 3 Portability (%): 88

R219 Lines: 100 Scores: 4 Portability (%): 88

R220 Lines: 161 Scores: 6 Portability (%): 88

R221 Lines: 55 Scores: 2 Portability (%): 89

R222 Lines: 110 Scores: 4 Portability (%): 89

R223 Lines: 174 Scores: 6 Portability (%): 89

R224 Lines: 278 Scores: 10 Portability (%): 89

R225 Lines: 310 Scores: 11 Portability (%): 89

R226 Lines: 415 Scores: 15 Portability (%): 89

R227 Lines: 94 Scores: 3 Portability (%): 90

R228 Lines: 81 Scores: 2 Portability (%): 92

R229 Lines: 466 Scores: 12 Portability (%): 92

R230 Lines: 92 Scores: 2 Portability (%): 93

R231 Lines: 95 Scores: 2 Portability (%): 93

R232 Lines: 161 Scores: 2 Portability (%): 96

R233 Lines: 184 Scores: 2 Portability (%): 96

R234 Lines: 0 Scores: 0 Portability (%): 100

R235 Lines: 0 Scores: 0 Portability (%): 100

R236 Lines: 19 Scores: 0 Portability (%): 100

R237 Lines: 19 Scores: 0 Portability (%): 100

R238 Lines: 19 Scores: 0 Portability (%): 100

R239 Lines: 19 Scores: 0 Portability (%): 100

R240 Lines: 19 Scores: 0 Portability (%): 100

R241 Lines: 19 Scores: 0 Portability (%): 100

R242 Lines: 19 Scores: 0 Portability (%): 100

R243	Lines:	19	Scores:	0	Portability (%):	100
R244	Lines:	19	Scores:	0	Portability (%):	100
R245	Lines:	19	Scores:	0	Portability (%):	100
R246	Lines:	19	Scores:	0	Portability (%):	100
R247	Lines:	19	Scores:	0	Portability (%):	100
R248	Lines:	19	Scores:	0	Portability (%):	100
R249	Lines:	19	Scores:	0	Portability (%):	100
R250	Lines:	19	Scores:	0	Portability (%):	100
R251	Lines:	19	Scores:	0	Portability (%):	100
R252	Lines:	19	Scores:	0	Portability (%):	100
R253	Lines:	21	Scores:	0	Portability (%):	100
R254	Lines:	21	Scores:	0	Portability (%):	100
R255	Lines:	21	Scores:	0	Portability (%):	100
R256	Lines:	21	Scores:	0	Portability (%):	100
R257	Lines:	23	Scores:	0	Portability (%):	100
R258	Lines:	23	Scores:	0	Portability (%):	100
R259	Lines:	35	Scores:	0	Portability (%):	100
R260	Lines:	35	Scores:	0	Portability (%):	100
R261	Lines:	40	Scores:	0	Portability (%):	100
R262	Lines:	44	Scores:	0	Portability (%):	100
R263	Lines:	45	Scores:	0	Portability (%):	100
R264	Lines:	52	Scores:	0	Portability (%):	100
R265	Lines:	52	Scores:	0	Portability (%):	100
R266	Lines:	53	Scores:	0	Portability (%):	100
R267	Lines:	55	Scores:	0	Portability (%):	100
R268	Lines:	70	Scores:	0	Portability (%):	100
R269	Lines:	77	Scores:	0	Portability (%):	100
R270	Lines:	80	Scores:	0	Portability (%):	100
R271	Lines:	81	Scores:	0	Portability (%):	100
R272	Lines:	91	Scores:	0	Portability (%):	100
R273	Lines:	99	Scores:	0	Portability (%):	100
R274	Lines:	101	Scores:	0	Portability (%):	100
R275	Lines:	101	Scores:	0	Portability (%):	100
R276	Lines:	226	Scores:	0	Portability (%):	100

```
TOTAL NUMBER OF UNITS   122
SCORE FOR TOTAL UNITS   236
TOTAL NUMBER OF LINES   37873
FINAL ACCUMULATED SCORE   4823
VALUE OF WEIGHTING CONSTANT     3.00000

TOTAL PORTABILITY      61.80%
```

Appendix 3. Examples of Ingres Database Tables

```
1> select * from files

+------------------------+
|file                    |
+------------------------+
|nomisdata/R1            |
|nomisdata/R2            |
|nomisdata/R3            |
+------------------------+
(276 rows)


2> select * from master

+----------------+------------+--------+------------+------------+
|file            |lines       |hexmem  |memory      |subfunc     >
+----------------+------------+--------+------------+------------+
|nomisdata/R53   |          45|000006A8|            |           1>
|nomisdata/R152  |          50|000039C8|            |           1>
+----------------+------------+--------+------------+------------+

+----------+------------+------------+------------+------------+
<extrefs   |mtsroutines |cinequiv    |cincommon   |varlen      >
+----------+------------+------------+------------+------------+
<         0|           0|           0|           0|           0>
<         0|           0|           0|           0|           0>
+----------+------------+------------+------------+------------+

+----------+------------+------------+------------+------------+
<funclen   |longvar     |dataorder   |hexdata     |zformat     >
+----------+------------+------------+------------+------------+
<         0|           0|           0|           0|           0>
<         0|           0|           0|           0|           0>
+----------+------------+------------+------------+------------+
```

```
+----------+------------+------------+------------+------------+
<qformat   |hollerith   |ampersand   |continuewith |datainit    >
+----------+------------+------------+------------+------------+
<         0|           0|           0|           0|           0>
<         0|           0|           0|           0|           0>
+----------+------------+------------+------------+------------+
+------------+------------+------------+----------+------------+
<dollarinvar |statements  |units       |unit5     |unit6       >
+------------+------------+------------+----------+------------+
<           0|           5|           0|         0|           0>
<           0|           5|           0|         0|           0>
+------------+------------+------------+----------+------------+
+------------+-----------+------------+------------+----------+
<intdeffiles |quotes     |doloops     |freeform    |longlines >
+------------+-----------+------------+------------+----------+
<           0|          0|           3|           0|         0>
<           0|          0|           1|           0|         0>
+------------+-----------+------------+------------+----------+
+------------+
<ascii       |
+------------+
<           1|
<           1|
+------------+
(276 rows)

2> select * from subroutines

+------------------------+--------------+
|file                    |routine       |
+------------------------+--------------+
|nomisdata/R53           |ROUTINE53     |
|nomisdata/R152          |ROUTINE152    |
+------------------------+--------------+
(616 rows)
```

```
1> select * from extrefs

+-------------------------+---------------+
|file                     |extref         |
+-------------------------+---------------+
|nomisdata/R106           |OPENC          |
|nomisdata/R173           |ANAMES         |
|nomisdata/R173           |CHAINR         |
+-------------------------+---------------+
(1318 rows)

2> select * from mtsroutines

+-------------------------+---------------+
|file                     |mtsroutine     |
+-------------------------+---------------+
|nomisdata/R7             |SYSTEM         |
|nomisdata/R7             |WRITE          |
|nomisdata/R8             |SYSTEM         |
|nomisdata/R8             |WRITE          |
+-------------------------+---------------+
(75 rows)

2> select * from criteria

+---------------+-------------+
|criterion      |critcount    |
+---------------+-------------+
|cinequiv       |            4|
|cincommon      |            3|
|varlen         |            4|
+---------------+-------------+
(31 rows)
```

```
2> select * from mtssubr
```

mtsroutine	mtscount
ADROF	5
ATNTRP	6
CMD	8

(27 rows)

```
2> select * from scores
```

file	score	lines	portability
nomisdata/R53	8	45	46
nomisdata/R153	4	50	76

(276 rows)

References

BLAK89 BLAKEMORE, M. & NELSON, R. (1989) Data compaction in NOMIS, a geographic information system for the management of employment, unemployment and population data. *University Computing* 7: 144-147.

COMP90 COMPUTER WEEKLY (1990) Battle Rages on at the Fortran Front. *Computer Weekly* May 1990.

COWE77 COWELL, W. (1977) Portability of Numerical Software. in *Lecture Notes in Computer Science* No. 57, Introduction Goos, G. & Hartmanis, J. (eds.): Springer-Verlag, New York.

DATE87 DATE, C. J. (1987) *A Guide to INGRES.* Addison Wesley.

HAGU76 HAGUE, S. J. & FORD, B. (1976) Portability - Prediction and Correction. *Software - Practice and Experience* **6**: 61-69.

HATT88 HATTON, L., WRIGHT, A., SMITH, S., PARKES, G., BENNET, P. & LAWS, P. (1988) The Seismic Kernel System - A Large-Scale Exercise in Fortran 77 Portability. *Software - Practice and Experience* **18**: 301-329.

HUNT90 HUNTER, G. (1990) The Fate of Fortran-8x. *Communications of the ACM* **33**: 389-391.

LARM81 LARMOUTH, J. (1981) Fortran 77 Portability *Software - Practice and Experience* **11**: 1071-1117.

LEMO81 LEMOINE, M. & MULLOR, J. (1981) Software Transferability: A Practical Approach. *Software - Practice and Experience* **11**: 425-433.

MOON90 MOONEY, J. D. (1990) Strategies for Supporting Application Portability. *Computer* **November 1990**: 59-70.

REIN77 REINSCH, C. (1977) Some Side Effects of Striving for Portability. in *Lecture Notes in Computer Science* No. 57, pp 3-19 Goos, G. & Hartmanis, J. (eds.): Springer-Verlag, New York.

SEAC90 SEACORD, R. C. (1990) User interface management systems and application portability. *Computer* **October 1990**: 73-75.

SOMM92 SOMMERVILLE, I. (1992) *Software Engineering.* Addison-Wesley.

SMIT77 SMITH, B. T. (1977) Fortran Poisoning and Antidotes. in *Lecture Notes in Computer Science* No. 57, pp 178-256 Goos, G. & Hartmanis, J. (eds.): Springer-Verlag, New York.

TANA92 TANAKA, M. (1992) A Study of Portability Problems and Evaluation. *Railway Technical Research Institute, Japan* Technical Report: 90-95.

TANE78 TANENBAUM, A. S., KLINT, P. & BOHM, W. (1978) Guidelines for Software Portability. *Software - Practice and Experience* 8: 681-698.

WALL82 WALLIS, P. J. L. (1982) *Portable Programming.* Macmillan.

Bibliography

AHO, A. V., KERNIGHAN, B. W. & WEINBERGER, P. J. (1988) *The Awk programming language.* Addison Wesley.

BAILEY, D. H. (1990) In Response to The Fate of Fortran-8x. *Communications of the ACM* **33**: 391-392.

BRAINERD, W. (1990) The Fate of Fortran-8x - Additional Thoughts. *Communications of the ACM* **33**: 392.

BROWN, P. J. (1972) Levels of Language for Portable Software. *Communications of the ACM* **15**: 1059-1062.

ELLIS, T. M. R. (1982) *A structured Approach to Fortran 77 Programming.* Addison Wesley.

GARDNER, D. R. S. (1986) An Investigation into the style of Pascal Programming. *Computer Science Report, Durham University.*

SHEARING, G. (1989) Conversion of MTS Fortran programs to Standard Fortran 77. *NUMAC Documentation.*