Graduate Theses, Dissertations, and Problem Reports

2004

Developing the portability indices and the portable code generator

Gregory E. Mundy
West Virginia University

Follow this and additional works at: <https://researchrepository.wvu.edu/etd>

Recommended Citation

Mundy, Gregory E., "Developing the portability indices and the portable code generator" (2004). *Graduate Theses, Dissertations, and Problem Reports*. 1503.
<https://researchrepository.wvu.edu/etd/1503>

This Thesis is protected by copyright and/or related rights. It has been brought to you by the The Research Repository @ WVU with permission from the rights-holder(s). You are free to use this Thesis in any way that is permitted by the copyright and related rights legislation that applies to your use. For other uses you must obtain permission from the rights-holder(s) directly, unless additional rights are indicated by a Creative Commons license in the record and/ or on the work itself. This Thesis has been accepted for inclusion in WVU Graduate Theses, Dissertations, and Problem Reports collection by an authorized administrator of The Research Repository @ WVU. For more information, please contact researchrepository@mail.wvu.edu.

DEVELOPING THE PORTABILITY INDICES AND THE PORTABLE CODE GENERATOR

Gregory E. Mundy

Thesis submitted to the
College of Mineral and Engineering Sciences
at West Virginia University
in partial fulfillment of the requirements
for the degree of

Master of Science
in
Computer Science

James Mooney, Ph.D., Chair
John Atkins, Ph.D.
Cynthia Tanner, MSc.

Lane Department of Computer Science and Electrical Engineering
Morgantown, West Virginia
2004

Keywords: Software Portability, Metrics, Grammatical Evolution, Automatic
Code Generation, Code Abstraction

©2004 Gregory E. Mundy

Abstract

Developing the Portability Indices and the Portable Code Generator

Gregory Mundy

Portability is recognized as a highly desirable characteristic of contemporary software design. However, research in the area has not been forthcoming in recent years and the body of work related to the field has tended to be ad hoc and speculative at best. Software developers oftentimes fail to consider portability as a part of their overall system designs because the cost of integrating it into the software production process may be prohibitively high in the short term and it is sometimes difficult to justify to management these short-term overheads with respect to the long-term benefits to be gained. Furthermore, developers often overlook portability altogether, arguing that the standardized nature of popular programming languages such as Java, C, and C++ eliminate portability issues altogether, a notion that has proved to be false in many cases.

This study attempted to establish a framework for deriving a probabilistic measure of software portability that may be potentially useful in aiding developers to make key design decisions that will result in some quantifiable level of portability while allowing them to minimize the cost of incorporating portable design into the software production process. In essence, this methodology will allow developers to choose a subset of portability practices they wish to incorporate into their design and estimate what degree of portability may be achieved by adopting these practices. In order to measure the degree of portability that may result from the design decisions, the framework relies on two metrics, namely the Software Portability Index (SPI) and the Environment Portability Index (EPI).

A secondary goal of this research was to explore the potential portability benefits that may be derived from applying evolutionary techniques, specifically the concept of Grammatical Evolution [O'N99a], to the design of a specialized tool that is capable of analyzing source code modules for potential portability issues and then regenerating more portable versions of these modules in a programming language that may or may not be the same that the original program was written in. This research resulted in the development of a rudimentary tool, referred to as the Portable Code Generator (PCG), capable of detecting a small subset of common portability problems in C++ source code written with a limited range of language features. In addition, the Portable Code Generator was able to convert, with varying degrees of success, the original source code to Perl, C as well as a more portable C++ implementation that eliminated the portability issues detected in the original code.

Acknowledgements

During the production of this work, I have learned a great deal about my abilities to produce quality research work that spans a wide variety of areas in Computer Science. However, without the support and guidance of so many people, completion of this work would have been impossible. First, I would like to thank God for giving me the strength and the perseverance to complete this work. I would like to thank my research advisor, Dr. James Mooney, a master in the field of software portability, and an all around nice guy, for his kind support, guidance, and encouragement of my ideas. Dr. Mooney, I know that when I started working on this thesis, my mind was a disorganized clutter of a million different ideas. Thank you for your patience in helping me to define the scope of my project into something realistic and achievable.

I would also like to acknowledge my committee, Dr. John Atkins and Mrs. Cynthia Tanner. Thank you both for believing in my ability to produce research worthy of a Masters degree. Mrs. Tanner, I will never forget you as my first Computer Science instructor at West Virginia University. You taught me how to analyze problems and establish solutions before attempting to write code, a practice I still adhere to even today. Dr. Atkins, I thank you for being a great initial advisor and guiding me through my early days of Grad School.

To my father, Mr. Cecil E. Mundy, thank you for being a constant source of advice and encouragement to me. To my aunt Winsome, I admire you for your academic and personal accomplishments. You've taught me a great deal about what it takes to be a good Computer Scientist and a great person, I only hope that I can live up to the high standards you've established. To my aunt Jackie and my uncle Carl, thank you both for being supportive of my endeavors and always encouraging me to pursue my dreams. To my uncle Colin, thank you for helping me to overcome my feelings of hopelessness when things weren't going my way, you've been a steadfast source of inspiration to me. To all my family members who have contributed to making me a success, in one way or another, I thank you.

To my girlfriend, Ms. Maria Isabel Charle (Maribel), for her constant support and encouragement of my work. Maribel, I know that it must have been very difficult for you to sit down and listen to me rant about my ideas and offer suggestions, but I thank you for your patience and understanding.

To Dr. Ray Morehead for providing me with funding as a Teaching Assistant throughout my graduate student career and to the administrative staff: Lucy, Maggie and Linda, for navigating me through the treacherous waters of forms, deadlines and deliverables!

I'd like to thank the wonderful friends, classmates, co-workers, students and fellow computer scientists I've met during my time at WVU: Andres, Rajesh, Vani, Dave Sh, Chris, Rex, Dave Sm, John, Patrick, Julie, Bruce, Brie.

Dedication

This thesis is dedicated to my grandparents, Mr. Cecil H. Mundy and Mrs. Adriana R. Mundy. One of the deepest regrets of my life is that my grandparents did not live to see the completion of my Masters degree. I am deeply indebted to them for instilling in me the values of hard work, persistence, and integrity. Without their early guidance, I would never have made it this far in my academic and personal life.

To my father, Mr. Cecil E. Mundy, who has constantly supported my decisions in life, even though I know now that some of my decisions were questionable in the earlier years! I thank you for constantly offering encouragement and support to me during the course of my studies. I appreciate everything that you have done and continue to do for me. You are truly the world's greatest Dad! I love you.

Contents

1	Introduction	1
1.1	Definitions of Software Portability	4
1.2	The Portability Indices	6
1.2.1	Software Portability Index	6
1.2.2	Environment Portability Index	7
1.2.3	Relating the SPI to the EPI	8
1.3	The Portable Code Generator	9
1.4	Organization	10
2	Literature Review	12
2.1	Software Portability	12
2.1.1	Current Portability Research	12
2.1.2	Software Portability Metrics	17
2.2	Grammatical Evolution	18
2.2.1	Introduction to Grammatical Evolution	18
2.2.2	Practical Applications of Grammatical Evolution	20
2.3	Automatic Programming	21
2.3.1	Overview of Automatic Programming	21
2.3.2	Applications of Automatic Programming to Portability	22
2.4	Summary	24
3	A Probabilistic Measure of Portability: The Portability Indices	26
3.1	The Software Portability Index Checklist	26
3.1.1	Software Lifecycle	27
3.1.2	Programming Language	30
3.1.3	Operating System	31
3.1.4	Hardware Architecture	32
3.2	Computing the Software Portability Index	32
3.3	The Environment Portability Index Checklist	33
3.3.1	Operating System	34
3.3.2	Programming Language	34
3.3.3	Hardware Architecture	35
3.4	Computing the Environment Portability Index	35
3.5	The Portability Index Threshold	36

3.6	Interpreting The Portability Indices	37
3.7	Weighted Attributes	37
3.8	Further Considerations	40
4	Portability Index Case Studies	42
4.1	Case Studies	43
4.1.1	Case Study 1: Video Game	43
4.1.2	Case Study 2: Special Purpose Compiler	47
4.2	Merit of the Portability Indices	50
5	The Portable Code Generator	54
5.1	Architecture of the Portable Code Generator	54
5.1.1	The Code Analyzer	57
5.1.2	The Code Abstraction Unit	59
5.1.3	The Code Generator	61
5.1.4	The Code Tester	62
5.2	Further Considerations	63
6	Applications of the Portable Code Generator	64
6.1	Implementation Language	65
6.2	Caveats	65
6.3	Training the Portable Code Generator	66
6.4	Resolution of C++ Portability Problems	68
6.5	Applying the Portable Code Generator	70
6.6	Summary	71
7	Summary and Conclusion	72
7.1	Summary and Conclusion	72
7.2	Future Work	73
A	Portability Index Estimation Tool	75
A.1	Overview	76
A.2	Usage Instructions	76
A.2.1	Execution	76
A.3	Source Code Listing	76
B	Portable Code Generator	82
B.1	Overview	83
B.2	Usage Instructions	83
B.2.1	Files Created	83
B.3	Source Code Listing	83
C	Results from Portable Code Generator	98
C.1	train1.cpp	99
C.1.1	Original C++ Code	99
C.1.2	Abstract Code	99
C.1.3	Re-generated C Code	99

	C.1.4	Re-generated Perl Code	99
	C.1.5	Re-generated C++ Code	99
C.2	train2.cpp		100
	C.2.1	Original C++ Code	100
	C.2.2	Abstract Code	100
	C.2.3	Re-generated C Code	100
	C.2.4	Re-generated Perl Code	101
	C.2.5	Re-generated C++ Code	101
C.3	train3.cpp		101
	C.3.1	Original C++ Code	101
	C.3.2	Abstract Code	102
	C.3.3	Re-generated C Code	102
	C.3.4	Re-generated Perl Code	102
	C.3.5	Re-generated C++ Code	103
C.4	factorial.cpp		103
	C.4.1	Original C++ Code	103
	C.4.2	Abstract Code	104
	C.4.3	Re-generated C Code	104
	C.4.4	Re-generated Perl Code	104
	C.4.5	Re-generated C++ Code	105
C.5	arraydemo.cpp		105
	C.5.1	Original C++ Code	105
	C.5.2	Abstract Code	107
	C.5.3	Re-generated C Code	108
	C.5.4	Re-generated Perl Code	109
	C.5.5	Re-generated C++ Code	110
C.6	binary.cpp		111
	C.6.1	Original C++ Code	111
	C.6.2	Abstract Code	111
	C.6.3	Re-generated C Code	112
	C.6.4	Re-generated Perl Code	112
	C.6.5	Re-generated C++ Code	112

List of Figures

2.1	Traditional Software Process v. MetaH Software Process [Lew02]	15
5.1	Portable Code Generator Flowchart	56
6.1	train1.cpp: Initial Training Program	67
6.2	train2.cpp: Second Training Program	67
6.3	train3.cpp: Final Training Program	68

List of Tables

3.1	Portability Indices Rating Scale	32
3.2	Portability Indices Attribute Weighting Table	38
3.3	Weights assigned to SPI Attributes	39
3.4	Weights assigned to EPI Attributes	39
4.1	Video Game SPI Checklist (No Portable Design Considered) . . .	44
4.2	Video Game EPI Checklist	45
4.3	Average of Video Game SPI and EPI (No Portable Design Considered)	46
4.4	Video Game SPI Checklist (Portable Design Considered)	51
4.5	Average of Video Game SPI and EPI (Portable Design Considered)	51
4.6	Specialized Compiler SPI Checklist (No Portable Design Considered)	52
4.7	Specialized Compiler EPI Checklist	52
4.8	Average of Specialized Compiler SPI and EPI (No Portable Design Considered)	53
4.9	Specialized Compiler SPI Rating Scale (Portable Design Considered)	53
4.10	Average of Specialized Compiler SPI and EPI (Portable Design Considered)	53

Chapter 1

Introduction

In 1965 Gordon Moore observed that the number of transistors per square inch on integrated circuits had doubled every year since the invention of the integrated circuit in 1958 [Moo65]. This observation, coupled with his prediction that this doubling of integrated circuit capacity would endure over many decades, led to what is now known as *Moore's Law*. Moore's Law implied that, with the high rate of increase of integrated circuit capacity, computer hardware would rapidly evolve in terms of speed, capacity, processing power, and so on.

Although Moore's Law is nearly 40 years old, hardware still obeys it (with newer, faster hardware appearing on an average of 18 months). Hardware developers, in order to maintain Moore's Law, have oftentimes had to adopt technologies and design strategies that cause newer architectures to radically differ from their predecessors. In addition, the relatively low cost of hardware makes newer technologies highly accessible to consumers who, for a variety of reasons, demand more advanced hardware as soon as it becomes available.

The rapid increase in hardware technology coupled with the high demand for more powerful products by consumers creates a vibrant marketplace for the computing industry. However, consumers also demand that the software that ran on their old hardware should also run on the newer hardware as well.

This requirement for "forward-compatibility" is often quite difficult for software developers to achieve, particularly when a software application may be heavily dependent upon certain characteristics of the older architecture that may not exist in the new one.

The highly intangible, discrete nature of software makes it quite difficult to determine how well, if at all, a software application will be able to run on hardware that may not have existed when it was written. In many cases, software must be heavily modified or completely redeveloped in order to run on new hardware platforms. The concept of software portability evolved as a means of addressing the high cost issues associated with constantly modifying or redeveloping software for new architectures.

Although the evolution of new hardware plays an important role in portability considerations, it is not the only issue that portability addresses. In the context of this discussion, the term new hardware platform can be generalized to mean any platform that the software is not currently implemented on. Additionally, portability must consider a holistic view of the computing environment, which includes all the hardware, software, documentation, and users that describe and/or interact with the software. Portability must be concerned with changes in any and all of these variables that constitute the environment that the software is a part of and, as such, must address issues such as changes in operating systems, new user requirements, and so on.

The cost to modify or re-write existing software to support changes in an environment may be prohibitively high and generally requires large investments in time, money, and other resources. The concept of software portability stemmed from the need to reduce the cost of targeting software applications to different environments. The goal of portable design is to incorporate a variety of strategies into the software development lifecycle that have portability as a prominent

goal, thereby facilitating an easier transition for existing software to new environments, while reducing the cost of redeveloping the software.

Software portability is very important with respect to reducing the cost of producing software that runs on multiple platforms; however, portability results in several additional issues that must be considered, such as:

- The cost of incorporating portability into software design may be extremely high in the short term, a fact which may be difficult to justify to management.
- Portable design oftentimes leads to software that may perform poorly in a new environment because the design may not incorporate architecture specific optimizations which help to improve the performance of the software, as these optimizations may result in low portability.
- From an implementation standpoint, it is quite difficult to determine what portability issues may occur within the syntactic and semantic structure of a specific programming language without thorough knowledge of the programming language standard (if one exists).

This research will focus on outlining two metrics, referred to as the *Software Portability Index* and the *Environment Portability Index*, that will aid software developers in choosing what aspects of portable design they wish to incorporate into the overall design of the system, with the goal of minimizing the cost of portability and establishing some measurable value for the degree of portability that may be achieved as a result of their design choices. It is intended for the indices to be useful as a negotiation tool between developers and managers in order to achieve some consensus between the short-term cost overheads versus the long-term portability goals. In this text, the Software Portability Index and the Environment Portability Index will be collectively referred to as the

Portability Indices.

This research will also explore the concept of automatic code generation and determine how the concept may be used to aid portability efforts. A rudimentary portability code analyzer/generation tool, inspired by the concept of Grammatical Evolution [O’N99a], referred to as the *Portable Code Generator*, or PCG, will be developed as a part of this exploration process. Ultimately, it is hoped that the PCG will be able to help developers with the implementation considerations of portability by testing code for known portability problems (based on real-world experience) and potentially re-generating the code into a more portable version that may or may not be written in the same programming language as the original. The remainder of this chapter will focus on establishing several definitions of portability as well as providing a brief discussion of the Portability Indices and the Portable Code Generator.

1.1 Definitions of Software Portability

[Moo97] defines porting as:

the act of producing an executable version of a software unit or system in a new environment, based on an existing version.

In addition, [Moo97] defines portability as:

A software unit is portable (exhibits portability) across a class of environments to the degree that the cost to transport and adapt it to a new environment in the class is less than the cost of redevelopment.

Mooney’s definitions of portability implies that portability may be viewed as the inverse of redevelopment based on the fact that portability is concerned with the degree of changes that need to be made to the existing software in order for it to operate in a new environment whereas redevelopment is interested in the ability

to construct a new software application, specifically for the new environment, that may be functionally equivalent to the original software.

Portability is oftentimes confused with reusability. It is true that these two concepts are very similar in nature; however, [Moo95] and [Pou94] offers the distinction that, while portability may be viewed as a special case of reusability, reusability as a practice is generally concerned with maintaining a collection of various components that can be used in a variety of environments, whereas portability deals with reusing complete software applications in a new environment. It can thus be argued from Mooney and Poulin's observations that portability subsumes reusability.

It is important to note that, although portability is regarded as a very important characteristic of modern software, unlike many other software engineering issues including reusability, there is a severe dearth of metrics that can be used to describe the effectiveness of portability practices. [Pou94] cites one of the earliest measures of portability used by the U.S. Department of Defense as simply counting the number of lines of code a programmer had to change in a particular software module to adapt it to the new environment and then using a mathematical function to convert the number of changed lines to a portability value. It is quite obvious that this measure does not hold any meaningful conclusions, particularly when considering the complexity of modules used to implement many software applications.

[Moo97] suggests a measure referred to as the *Degree of Portability* that is expressed simply as:

$$DP = 1 - (\text{cost to port} / \text{cost to redevelop})$$

The Degree of Portability measure is useful for developers because it offers some meaningful value that can be evaluated to determine when it is more feasible to redevelop a particular software application as opposed to porting it and vice

versa.

Another attempt at establishing some way of measuring and assessing portability is the Open Micro Initiative Software Portability Assessment Project, termed OMI/SPAM. According to [OMI95], the project was to last for approximately two years; however, there is no information available that tells how the methodology worked and whether or not it was successful.

1.2 The Portability Indices

The Portability Indices are intended to contribute to the world of software portability metrics. They represent an attempt at establishing a probabilistic measure of portability that allows software developers to choose what aspects of portable design they wish to incorporate into the software production process and provide an indication of how much effort will be required to port a particular software application to a new environment, based on the selected portability considerations incorporated into the product and the natural portability of the environment(s) being ported to. The Portability Indices will allow developers to establish some middle ground between the high costs of implementing portability strategies versus how much cost overheads management will be willing to accept in favor of portability.

1.2.1 Software Portability Index

The Software Portability Index is a reflection of the degree to which portability is considered as a part of the design of the original software. In order to compute the Software Portability Index, developers will be required to complete a *Software Portability Index Checklist* that is comprised of four categories, namely software lifecycle, programming language, operating system and hardware architecture. Each category in the Software Portability Index Checklist

is comprised of various attributes that attempt to identify the ways in which portability may be considered at varying levels of software design as discussed in [Moo97] and determining to what extent the developer will adhere to each portability consideration. For each portability attribute, the developer specifies a numeric value between 0 and 4 (where 0 is the lowest and 4 is the highest), which represents the extent to which the attribute will be adhered to during the development of the software.

The Software Portability Index is computed as the sum of all attributes in the Software Portability Index Checklist and normalized to a value between 0 and 1 by dividing the sum of attributes by the total possible value of the Software Portability Index Checklist. Individually, the Software Portability Index can be used to measure how important portability is perceived to be in the software being developed, with 0 representing no portability considerations and 1 representing the highest regard for portability. However, as will be described in later chapters, the Software Portability Index is intended for use in conjunction with the Environment Portability Index to provide some indication to developers as to how much effort will be required to port the software to a new environment, given the portability built into the software and the potential portability of the new environment.

1.2.2 Environment Portability Index

The Environment Portability Index is very similar to the Software Portability Index. However, it is intended to provide a (probabilistic) measure of the suitability of the environment to the software to be ported. In effect, the Environment Portability Index will provide a reflection of the difficulty of porting an application to the environment, which may be useful in facilitating a decision as to whether or not portability is a better option than redevelopment.

The computation of the Environment Portability Index is performed in a similar manner to the method used to derive the Software Portability Index. Developers must complete an *Environment Portability Index Checklist* that comprises of three sections, namely programming language, operating system, and hardware. It is noted that software lifecycle isn't considered in this checklist because the software already exists and the Software Portability Index has previously captured these considerations.

The Environment Portability Index Checklist employs the same attribute rating scale used by the Software Portability Index Checklist and the Environment Index is computed as the sum of the attributes in the Environment Portability Index Checklist and normalized to a value between 0 and 1 by dividing the sum by the total possible value for the Checklist. Unlike the Software Portability Index, the Environment Portability Index is meaningless if considered individually.

1.2.3 Relating the SPI to the EPI

The Software Portability Index and the Environment Portability Index are related by the *Portability Index Threshold*. The Portability Index Threshold is best viewed as a user-defined value that represent what a developer would consider to be "good" portability for a particular software application. Like the Portability Indices, the Portability Index Threshold ranges between 0 and 1.

The average of the Portability Indices are calculated and compared to the Portability Index Threshold. If the average is less than the Portability Index Thresholds, then it can be argued that porting is not necessarily the best option and the decision whether or not to port may be made based on the disparity between both values. If the average is greater than or equal to the Threshold, then porting can be considered a very good option and the potential difficulty

of port can be measured from the difference between the values. In addition, if the average is 1, it may be implied that perfect portability is possible.

The Portability Indices and how they may be applied are discussed in greater depth in Chapters 3 and 4.

1.3 The Portable Code Generator

The Portable Code Generator is a portable code analysis/generator tool that is inspired by the concept of Grammatical Evolution [O’N99a]. The purpose of the Portable Code Generator is to evaluate source code for potential portability issues based on real world portability experiences that have been documented in articles such as [DLKR00] and [Wil98]. In addition, the Portable Code Generator will attempt to represent the source code in a high-level machine independent representation that is then used to generate a functionally equivalent, portable version of the source code in a language that may or may not be the same as the original code.

The Grammatical Evolution component comes into play when the user performs acceptance tests on the new code and, if the code does not meet the desired requirements, the user may modify the machine independent representation and then repeat the regenerate, test, tweak cycle as often as necessary until the code that best meets the requirements is evolved.

Overall, the intent of the Portable Code Generator is to aid developers in writing more portable code much faster while the Portability Indices are intended to provide some a priori knowledge of potential success or failure of applying design decisions in favor of portability to the software production process and the impact these decisions have on the portability to new environments. Superficially, the two aspects of this research seem completely unrelated; however, they may be loosely coupled together by analyzing their purposes. Ideally,

the Portability Indices tell whether or not portability is more favorable with respect to redevelopment. The Portable Code Generator can be used to somewhat nudge the designer toward recognizing portability as a viable choice even when the Portability Indices indicate that redevelopment is the more viable option. [FF99] demonstrates this concept by showing how automatic code generation was able to convert software applications written in IBM 370 machine code to C code that could be run on a variety of different architectures.

A more detailed discussion of the Portable Code Generator will be provided in Chapters 5 and 6.

1.4 Organization

The remainder of this thesis will be organized as follows:

- Chapter 2 will be a literature review that will form the background for this research. The concepts of Portability, Automatic Programming, and Grammatical Evolution will be discussed in the context of contemporary research in the fields. In addition, several case studies that achieve portability by automated methods will be discussed in some depth.
- Chapter 3 will discuss the Portability Indices and provide a high level discussion of how these measures relate to other prominent measures for portability.
- Chapter 4 provides a series of case studies that demonstrate how the Portability Indices may be used in determining the portability of a software application. The intent of this chapter is to provide an informal validation of the approach.
- Chapter 5 will provide a description of the architecture of the Portable

Code Generator and determine how it will be used in the implementation phase of the software development lifecycle.

- Chapter 6 will discuss the results of the Portable Code Generator as applied to several example problems.
- Chapter 7 will be a final discussion and conclusion of this work and suggestions for how it can be extended in the future to be even more usable in the real world.

Chapter 2

Literature Review

This chapter will provide a review of previous research in the areas of Software Portability, Grammatical Evolution, and Automatic Code Generation, which together form the backdrop for the work conducted in this thesis. The chapter is intended to provide the reader with a basic background of each area by discussing a variety of concepts and case studies related to each field.

2.1 Software Portability

This section will provide a brief description of current research trends in the area of Software Portability. The availability of material that explicitly describes Portability is very limited and, as such, this section is a reasonable attempt to effectively identify and discuss the body of work associated with the field.

2.1.1 Current Portability Research

Software Portability is recognized as an important attribute of modern software design; however, there is not a great deal of current research that attempts to explain and quantify portability or find ways to improve it. [All02] contends that the concept of portability came about as a result of developers wanting to

be free from particular companies that manufactured proprietary hardware and compilers which effectively ensured that users were tied to a particular brand of architecture. [All02] further claims that introduction of languages such as C and the popularity of operating systems such as UNIX in the 70s and 80s aided in introducing portability.

The discussion presented in [All02] is interesting because it provides a view of portability from the Open Source community's perspective, where it is commonly believed that proprietary, "closed" software and hardware is not conducive to technological innovation. The article also highlights a very common belief that portability is an issue that is dealt with at the implementation (programming language and operating system) level. Similarly, articles such as [SG03] have termed portability as:

Relating to or being software that can run on two or more kinds of computers with two or more kinds of operating systems.

This definition once more isolates portability as an implementation ideal and further purports the misconception that portability issues can solely be solved by specific programming languages, such as C or Java, and standardized, open operating systems.

The positions presented in the papers previously discussed are not necessarily incorrect, however, they only describe a small segment of the portability issue. Dr. James Mooney is one of the leading researchers in the field of software portability and the results of his work assume a more holistic view of software portability. In [Moo97], Mooney describes an approach where portability is considered from the earliest design of the software product to the very end of the software's lifecycle. This holistic view proposed by Mooney is much more convincing when considering portability because similar articles, such as [Moo90] and [Moo93], have provided insight into how portability can be considered at all

phases of software development as well as discussing other concerns of portability (such as users, documentation, software interfaces, and cultural differences) that are not addressed by most other portability approaches. In essence, Mooney considers hardware architecture, operating system, and programming language to be a part of a larger *environment* that is made up of all aspects of the software and all entities that interact with or describe it. This environment includes everything from the software and hardware to users of the system and physical documents that describe the software.

[Moo02] offers further validation for Mooney’s holistic view of portability by describing the benefits that commercial software developers are able to gain by applying a portable design discipline to the development of their software, instead of simply finding arbitrary ways of porting their applications to new environments. In addition, [Moo02] describes how portability considerations can be applied to many different classes of computer systems, from personal computers to mainframe systems.

Portable design has the advantage of reducing dependencies between the software and the operating system and hardware architecture. However, this perceived advantage might become a serious prohibitive factor to software development depending on the nature of the application being developed. In the case of mission critical, real-time systems, the success of these applications relies on the dependencies between the software, hardware, and operating system. As such, a great deal of portability might not be achievable in these cases. [Lew02] offers a novel approach to the rapid development of portable, real-time applications by introducing the MetaH framework used to develop software for military devices, such as missile and aircraft flight systems.

[Lew02] describes MetaH as a language and a set of tools used to analyze and implement embedded real-time systems.

Figure 2.1: Traditional Software Process v. MetaH Software Process [Lew02]

Figure 2.1 provides a comparison between the traditional software production process and the MetaH software process. The major differences between both processes is that the traditional process is concerned with the specification of requirements and design on paper and then implement the design as a separate set of activities, while the MetaH methodology is based on a more flexible approach where modeling techniques are used in the requirements phase to describe various mission critical dependencies, such as reliability, fault handling, and safety and security. Additionally, the MetaH methodology incorporates incremental, automated integration of the system, as compared to the static implementation of the traditional process.

In effect, MetaH is a dynamic methodology that reduces communication barriers associated with conventional software development and allows for a better way of capturing and implementing requirements. Another important characteristic of the MetaH framework is that it utilizes the Ada95 language for code generation. Ada has long been known to provide a very high degree of portability due largely in part to the strictness of adherence to its standards that must be followed by all implementations of the language. In addition, the very versatile nature of the language has made it very popular for military applications and embedded-system applications. Therefore, it comes as no great surprise,

that the MetaH framework owes a large portion of its portability success to Ada95.

In addition to military applications, the field of High Performance Computing in general have benefited from portability considerations. [TWN98] highlights some valuable experience gained from portability efforts in High Performance Computing environments. The paper focuses mainly on providing "best practices" for achieving portability in High Performance systems. Interestingly, the authors go beyond the implementation aspects of portability and describe various principles and practices that developers may incorporate into their design. For instance, the authors provide a brief highlight of the importance of using a consistent programming style when writing portable code and possible measures, such as providing a coding procedure manual that could be taken to ensure the guidelines are followed.

[TWN98] is a unique article in the sense that it is the only one found in the entire course of this research that explicitly addressed the requirement of portability in an area of application as popular as High-Performance Computing.

One of the most recent phenomena in software development is the introduction of the concept of the Software Product Line [FV03]. The Software Product line attempts to mimic the functionality of an automated factory, such as an automobile production plant, that relies heavily on mechanization to build its products. Based on a brief review of the literature available on the subject, it is implied that the software product line subsumes portability and reusability. However, the core of Software Engineering practices is based on the fact that software is discrete and intangible and, as a result, it is very difficult to apply traditional engineering principles to it. This is not to say that the Software Product Line is not a novel approach, however, improving the portability discipline may help this concept along even further.

2.1.2 Software Portability Metrics

Most aspects of software development have a large number of metrics to describe and quantify their effectiveness. Software Portability, on the other hand, has very limited metrics to aid in this endeavor. [Pou94] makes brief mention of an early measure proposed by the U.S. Department of Defense that was to count the number of lines of source code that a programmer had changed to get a specific module to run in a new environment and then having a mathematical function map the ratio of number of lines changed to number of lines of code in the module to some portability value. This measure, though somewhat novel, was never incorporated due to the fact that it did not provide any conclusive information about the portability of an application.

Another attempt at establishing some measure of portability is found in the Open Micro Initiative Software Portability Assessment Methodology [OMI95]. OMI/SPAM, as it was referred to, was intended to provide a methodology for standardizing a great deal of the hardware and software architectures in Europe by providing some way of measuring compliance to guidelines established by the OMI. Unfortunately, nothing is known about the success or failure of this project, or whether or not it was even completed.

The most reasonable measure of Portability currently available is Mooney's Degree of Portability [Moo97]. As mentioned in Chapter 1, this measure provides a ratio between the cost of porting a software application and the cost of redeveloping the same application. Mathematically, the Degree of Portability is expressed as:

$$DP = 1 - (C_{port}/C_{redevelop}) \quad (2.1)$$

Where C_{port} is the perceived cost associated with porting the application to the new environment and $C_{redevelop}$ is the estimated cost of redeveloping the application for the new environment. Ideally, if the Degree of Portability evaluates to

a negative value, then it may be concluded that it is much cheaper to redevelop the software than to port it; however, if the Degree of Portability evaluates to a positive value, then it is reasonable to believe that it is more cost effective to port the application and the degree of benefit to be gained by porting is determined by the value of the Degree of Portability, where 1 implies perfect portability. Additionally, a Degree of Portability value of 0 indicates that the cost to port is the same as the cost to redevelop and it makes no difference as to which alternative is chosen. This final condition is extremely rare, or even impossible, because of the fact that there will always be some disparity between the cost of redevelopment and the cost of porting.

Mooney's Degree of Portability represents a logical approach to measuring portability, but it might be quite a difficult task to establish tangible values for the costs of portability and redevelopment. Provided that these costs can be accurately determined, this measure will provide a very good indicator to developers as to whether or not they should focus their energy on redeveloping or porting a software application.

2.2 Grammatical Evolution

This section will focus on providing a background for the evolutionary algorithm known as Grammatical Evolution. It is intended to provide the reader with a very high-level overview of the field and discuss some of the current research being conducted in this area.

2.2.1 Introduction to Grammatical Evolution

In 1959, Dr. Arthur Samuel put forward a simple question: "how can computers be made to do what needs to be done, without being told exactly how to do it?" For decades, this simple question has fuelled the imagination of many

Computer Scientists, leading to many interesting innovations that have sought to realize the goal of automatically generating programs from high-level problem statements, or specifications. Fields such as Generative Programming, Metaprogramming, and Intentional Programming [Hay03] are all examples of the varying approaches that have evolved with the goal of realizing the dream of what is nowadays generally referred to as Automatic Programming.

One of the most interesting innovations in the field of Automatic Programming is the Genetic Programming paradigm. Historically, Genetic Programming traces its roots back to the 1950's with experiments conducted by Turing, et al [Dem00]. However, Genetic Programming as a discipline first rose to prominence with Koza's landmark Hierarchical Genetic Algorithms study (circa 1989), which was an extension to Holland's Genetic Algorithm. The Hierarchical Genetic Algorithms concept was later refined (circa 1992) into what is now known as Genetic Programming [Dem00].

Genetic Programming may be best viewed as a "biologically-inspired, domain-independent method that automatically creates a computer program from a high-level statement of a problem's requirements." [K⁺99]. It incorporates the "Darwinian principles of reproduction and survival of the fittest and analogs of naturally occurring genetic operations such as crossover (sexual recombination) and mutation" [Koz95] to produce (or evolve) programs that solve, or approximately solve, a given problem.

Genetic Programming has proven to be a very useful tool in a variety of fields of study, including areas such as robotics, financial applications, pattern recognition, parallelization, and simulation and modeling [Koz97]. This field is still a very young one and there are many potential applications for this approach yet to be discovered.

Grammatical Evolution [O'N99b] represents a step up the Automatic Pro-

gramming evolutionary ladder. In principle, Grammatical Evolution is very similar to Genetic Programming, with the exception that it incorporates a Backus-Naur Form grammar that allows the algorithm to generate programs in any arbitrary language. The idea of combining a powerful search algorithm with the ability to generate source code in practically any language provides a great deal of potential for the future of Automatic Programming, such as the ability to evolve actual programs in any language that satisfies some given requirements.

2.2.2 Practical Applications of Grammatical Evolution

Currently, Grammatical Evolution is still in its infantile stages, but a great deal of success has been achieved with this powerful algorithm. One of the earliest successes of Grammatical Evolution was its ability to provide an efficient solution to the Santa Fe Trail problem as well as generate a very efficient caching algorithm for memory management [O’N99a].

In addition to the caching problem listed above, Grammatical Evolution has been applied to a variety of data mining and machine learning activities. For example, [O⁺01] outlines an application of Grammatical Evolution where trading rules for the Irish ISEQ stock market index were learned and applied to hypothetical investment situations. In general, the experiments were a relative success in light of the fact that the authors had just slightly over a year’s worth of data to analyze in the process of evolving the rules. In a hypothetical investment of \$10,000, it was found that the best trading rules generated yielded a \$5777 profit over the training period.

The field of Grammatical Evolution is a young one and, as a result, there is not yet a wide range of practical applications of this methodology as yet. Many researchers are becoming increasingly interested in this paradigm and

have begun to make many great contributions to the field. A list of projects and people working on Grammatical Evolution is available at the Grammatical Evolution research web site at <http://www.grammatical-evolution.org>.

2.3 Automatic Programming

In this final section, we discuss in some detail the fundamental ideas between Automatic Programming and how it relates to software development. Several case studies of the application of automated programming techniques using a variety of approaches and the portability benefits gained from these approaches are briefly discussed.

2.3.1 Overview of Automatic Programming

In the previous section that discussed Grammatical Evolution, brief mention was made to the idea of Automatic Programming. Evolutionary search algorithms, such as Genetic Programming and Grammatical Evolution, are just one aspect of the field. [CR93] outlines a variety of approaches to Automatic Programming that range from using high-level natural languages to represent programs, to logic-based program abstractions, and transformation models. The article provides readers with a good feel for the various directions that researchers take in finding ways to automatically generate code; however, there is no easy way to achieve this and programmers will still have to interact with a system in one way or the other to produce code. On the other hand, Automatic Programming makes software development more accessible to a wider range of users with varying degrees of technical abilities.

In [Har92], David Harel points out that model-based software development coupled with automatic code generation will be one of the major contributing factors to developing better software in a shorter space of time. The use of

model-based techniques in software development fosters the distinct advantage of being able to isolate the logic of the code being written from the actual implementation, which implies that portability could become a natural side effect of Automatic Programming, since all dependencies between hardware and software are eliminated. Similarly, [SUR96] provides valid arguments to support the claim that languages based on logic, such as Prolog, tend to be more portable than other language classes because they model the logic of what the program is supposed to do and leave the implementation details of how the tasks will be performed up to the low-level system.

Aside from the obvious advantage that logic-based programs have of abstracting logic from implementation, they provide a much better platform for code validation because it is much easier to verify logic than a group of interrelated procedures that may not have a very tangible flow of logic. On the down side, [SUR96] points out that there are a vast number of problems associated with using logic-based languages particularly in the fact that they are not widely used and oftentimes have problems handling very complicated I/O and GUIs.

Automatic Programming holds a great deal of potential for the future of software portability and, as the next section will demonstrate, there are several examples where Automatic Programming has helped in improving the viability of portability over redevelopment, even in the cases where the source and target environments have nothing in common and redevelopment seems like the most logical alternative.

2.3.2 Applications of Automatic Programming to Portability

The first case study that will be analyzed is found in [FF99] where the authors created a tool called Bogart that was able to convert programs originally written

in IBM 370 machine code to C for availability on a variety of other platforms. The success of Bogart is measured in the fact that it was able to convert several large modules that implemented a database management system and an application generator from IBM 370 native code to C. The authors compared their method of capturing the logic of the original code in an internal model and then applying transformations to the model to produce the new C code to a previous method of performing literal translations on each line of code.

The code produced by Bogart's, when compared to the literal translation tool, was determined to be more understandable, efficient, and portable than the code produced by literal translation. The fact that Bogart uses some form of a logical model to represent the structure of the program offers some support for the claims in [SUR96].

Another important example in favor of Automatic Programming is found in the PARAGEN project [WR95]. In this paper, Paul Walsh, one of the leading researchers in the field of evolutionary algorithms, and Conor Ryan, the founder of Grammatical Evolution, created a code generator called PARAGEN that used Genetic Programming techniques to convert serial code to parallel code. This approach is quite unique because of the fact that PARAGEN literally "grew" parallel code that was functionally equivalent to the serial code it was attempting to convert.

By adopting an evolutionary approach, Walsh and Ryan were able to successfully breed generations of programs and test how well they satisfied the requirements of the original program and reflected good parallel design. This approach entailed a cycle of generating a population of potential programs that addressed the problem, testing the population to determine the programs which demonstrated the best solution to the problem and/or the highest degree of parallelism (the fittest), and breeding a new generation of programs using the

”genes” of the best programs in the previous generation. This cycle was repeated until an optimal solution was found, or the number of generations exceeded some maximum value. The success of the PARAGEN system speaks volumes of the potential of simple evolutionary search algorithms being able to automatically breed populations of programs that solve some given set of requirements.

The final case study to be described in this paper is found in [Fra94] where the author, Michael Franz, describes a novel technique called Semantic Dictionary Encoding (SDE) that is used to provide an abstract representation of a given program. The fact that SDE was designed to use a very lightweight encoding scheme for programs, it is quite trivial to provide a code generator that is capable of providing very simple and efficient on-the-fly code generation on relatively slow processors without affecting the use of highly optimizing code generating methods on faster ones [Fra94].

The research described in [Fra94] has the potential of being able to provide a practical, universal software format which would lead to a high level of portability across a wide variety of heterogeneous architecture classes.

2.4 Summary

The review of literature provided in this chapter highlighted some of the more prominent research associated with the three disparate areas of Software Portability, Grammatical Evolution, and Automatic Programming that this thesis will attempt to unify into a metric and a tool that will provide some contribution to the field of Software Portability. It has become apparent from the research described in this chapter that many current software development practices attempt to subsume Portability. However, Portability of it self produces a set of unique challenges not addressed by many conventional software engineering principles. In short, there is need for much work in this field, particularly be-

cause the wide array of architectures and environments that exist in today's world are becoming more tightly coupled. Portability, regardless of whether or not it is termed code migration, software mobility, and so on, still needs to be studied in great depth to provide novel solutions that will aid in the tight integration of heterogeneous computing environments.

It was also observed from this review that there is a great deal of potential in the use of evolutionary search algorithms, such as Grammatical Evolution, which may yield very high benefits for finding solutions to the portable design versus performance tradeoffs that has become a prohibitive factor to incorporating Portability into conventional software design. It has been shown by the success of tools such as Bogart and PARAGEN that Automatic Programming is truly a field that should be closely tied to Software Portability because the techniques afford developers the opportunity to reap some portability benefit even when redevelopment is clearly a more reasonable option.

The remainder of this thesis will be focused on describing the Portability Indices metric that may be useful in augmenting Mooney's Degree of Portability measure and providing even more meaningful, quantifiable explanations for portability and the Portable Code Generator, a tool that draws its inspiration from Grammatical Evolution and Automatic Programming to provide similar benefits to Bogart and PARAGEN, with respect to breaking down the Portability barrier.

Chapter 3

A Probabilistic Measure of Portability: The Portability Indices

In Chapters 1 and 2 it was observed that there are very few metrics available to evaluate portable design. Outside of Mooney's Degree of Portability measure [Moo97] most other attempts at quantifying portability were either not informative to software developers or did not provide sufficient evidence to determine how useful the measure was. In this chapter, the Portability Indices and their potential impact on portable design is discussed. This chapter will form the basis for the case studies applying this methodology that will be described in Chapter 4.

3.1 The Software Portability Index Checklist

In Chapter 1, it was noted that the Software Portability Index is a reflection of the degree to which portability is considered as a part of the design of the original software. The Software Portability Index Checklist is the most essential component of the Software Portability Index because developers use it to specify the attribute values that are used to compute the Index. The Checklist is

comprised of four broad categories, namely:

1. Software Lifecycle
2. Programming Language
3. Operating System
4. Hardware Architecture

Each category is comprised of several attributes that will be discussed in further detail in subsequent sections. It must be noted that the attributes under consideration were mostly gathered from research described in [Moo93] with the exception of the Programming Language attributes that were collected from [How95].

The following sections will discuss each of the four main categories and the attributes that fall under each. It is important to note that the general convention for abbreviating each metric will be in the form:

ATTRIB

where ATTRIB is some meaningful abbreviation for the attribute.

3.1.1 Software Lifecycle

[Moo97] emphasizes that portability is not just an implementation-specific consideration. He argues that portability must be incorporated from beginning to end of the software production process. [Moo97] further claims that it is highly plausible to address portability issues from the very high level requirements elicitation process to the implementation phase. As such, this section concentrates on determining what steps the developer has taken in incorporating portability practices into the design aspect. The attributes of this category are loosely based on the waterfall methodology as described in [Moo97]. The sub-categories of this section include:

1. Specification
2. Design
3. Implementation
4. Testing and Debugging
5. Documentation
6. Maintenance

Specification

The Specification sub-category is concerned with determining what provisions were made to establish portability during the high-level requirements elicitation phase. The attributes associated with this sub-category are:

1. RED: Reduce system dependencies in requirements by determining how necessary apparent dependencies are.
2. PREQ: How much can portability can be considered as a requirement itself.
3. PLIM: Avoid portability limiting requirements.
4. PCOST: How much is one willing to pay for portability as a requirement.

Design

In the Design sub-category, the primary concern is to determine the extent to which the developer will adopt certain key design strategies, such as using formalized standards or adopting a suitable design strategy (such as Object-Oriented Design) that supports portability. The attributes in this sub-category are:

1. METH: Choose a suitable design methodology.
2. ISOL : Encapsulate (isolate) system dependant interfaces.
3. EXT : Identify external interfaces to each module.
4. STAND: Identify and design to suitable standards.

Implementation

The Implementation sub-category is primarily concerned with determining what degree of consistency in the implementation will be enforced for programmers. Note that this sub-category does not take into account the programming language(s) used in implementing the software because programming language choice is considered to be sufficiently important to have it's own category. The attributes in this sub-category are:

1. DISC: Use of a portability discipline.
2. IMPSTAND: Adherence to implementation standards.

Testing and Debugging

Testing is an important aspect of software development in general. The purpose of the Testing and Debugging sub-category is to determine how much the developer is willing to consider portability in testing and debugging. The attributes of this sub-category are:

1. PLAN: Use of portable test plans.
2. PREC: Maintain portable records of errors found and corrected.
3. WARN: Pay attention to compiler warnings.

Documentation

Documentation is an important characteristic of all software development. Based on observations in [Moo97], it is possible to make documentation portable in a variety of ways, such as isolating environment specific details so that they can be changed rapidly to support new environments. The attributes of this sub-category are:

1. SYSDEP: Separate system-dependent documentation.
2. CONF: Avoid confusing users.
3. PDOC: Document the porting process.

Maintenance

The Maintenance sub-category is the final one in the Software Lifecycle category. This sub-category provides some insight into how maintenance of the software application will obey a portability discipline. The following attributes are a part of this sub-category:

1. VER: Maintain single version of common parts of the software.
2. SIMP: Simplify modifications.

3.1.2 Programming Language

The choice of programming language used to implement a software application has a great impact on the portability of the software application itself. It is often found that languages, such as Ada, that have strict standardization policies generally tend to be highly portable because once a compiler or interpreter exists for the target platform it is guaranteed to work because the standards are rigidly enforced. On the downside, however, heavily standardized languages oftentimes

tend to be too inflexible and resistant to change to support innovations in the language. The goal of this section is to determine how flexible a programming language is with respect to supporting portability goals.

The following characteristics that must be considered in this section are:

1. LSTAND: Language is formally standardized.
2. LSUB: Language has a known "portable subset".
3. LENV: Ease of write compilers/interpreters for new environments.
4. LUND: Understandability of the language.
5. LMOD: Ability of the language to support modular design.

3.1.3 Operating System

The Operating System for which the original software application is written is another highly important portability issue that must be considered. Operating Systems that are implemented using well-defined standards are more likely candidates for portable design. It is natural to assume that a well-written operating system implemented in accordance with some widely known standard (such as POSIX) will be far better able to provide a level of abstraction that ensures that the software will not be able to distinguish between different hardware architectures.

The attributes of this category are:

1. OSSTAND: The Operating System is based on a formal standard.
2. OSABS: The level of abstraction from the hardware that the Operating System provides.
3. OSPORT: Portability of the Operating System.
4. OSOPEN: Is the Operating System implemented as open or closed.

3.1.4 Hardware Architecture

The hardware architecture that the software application is being implemented to run on is the final category of this checklist. It is quite difficult to ascertain what attributes are most meaningful for this category because it is noted that the Operating System and high level programming languages generally hide away the implementation-specific aspects of the hardware. The single attribute that is of most concern in this section is:

1. ARCSTAND: Degree of standardization of the hardware architecture.

3.2 Computing the Software Portability Index

In order to compute the Software Portability Index, a software developer must assign a value to each attribute in the Software Portability Index Checklist based on the rating scale described in Table 3.1.

Value	Definition
0	None
1	Low
2	Medium
3	High
4	Very High

Table 3.1: Portability Indices Rating Scale

Once all attributes are assigned a rating, the Software Portability Index is computed by Equation 3.1 where P_{attrib} refers to each attribute in the Software Portability Index Checklist and n refers to the total number of attributes in the Checklist. The value $S_{Portability}$ is a constant that represents the number of possible assignments for each attribute as listed in Table 3.1 and is multiplied by n to derive the maximum value of the Software Portability Index Checklist which is used to normalize the Software Portability Index to a value that ranges

between 0 and 1.

The Index may be interpreted in a similar manner to Mooney’s Degree of Portability [Moo97], where 0 implies low portability considerations and redevelopment will perhaps be the only alternative and 1 implies that all efforts have been made to ensure that the design is portable, therefore the chances of the software porting easily to a new environment is significantly high.

The Software Portability Index differs from Mooney’s approach in that it is meant to provide a speculative value for portability that *may* be achieved as a result of choosing between various competing design attributes, whereas the Degree of Portability provides some measure of the feasibility of portability over redevelopment given that there is some knowledge of the costs associated with portability and redevelopment.

$$SPI = \frac{\sum P_{attrib}}{S_{Portability} \sum n} \quad (3.1)$$

3.3 The Environment Portability Index Checklist

In order to provide further meaning to the Software Portability Index, it is necessary to compute the Environment Portability Index. Using both Indices it is possible for software developers to determine the difficulty of porting the software to a new platform. The Environment Portability Index Checklist is used to capture any a priori knowledge the developer may have regarding the new platform that the program is to be ported to.

The Environment Portability Index Checklist is made up of three major categories, namely:

1. Operating System

2. Programming Language

3. Hardware Architecture

Note that a category for Software Lifecycle is not included in the Checklist because the intent of portability is not to redevelop the software from the beginning, but rather to move it to a new environment with little or no modifications. It is assumed that any design decisions made in the original development of the software should be inherited by the ported versions. The next few sections will be dedicated to discussing each aspect of the Environment Portability Index Checklist.

3.3.1 Operating System

The Operating System is one of the most important issues that must be considered in the porting of software. It was discussed in previous sections that an Operating System that is implemented properly should generally try to shield software from the implementation specific details of the architecture. The following attributes that must be considered with respect to the Operating System are:

1. OSSIM: The Operating System for the target platform is the same (or similar) as the original operating system that the software was written for.
2. OSIMP: The target Operating System implements the same standards as the original.

3.3.2 Programming Language

The Programming Language is another important portability issue because if there are implementations of the language used to develop the original code

available on the new platform then it implies that there may be greater possibilities for portability (particularly if the language obeys some formal standards). The following attributes need to be considered for the Programming Language category:

1. LANGENV: The language that the application was originally written in is available in the new environment.
2. LANGIMP: The language conforms to formally established standards.

3.3.3 Hardware Architecture

As was the case with the Software Portability Index Checklist, it is difficult to determine the impact of hardware if the implementation is properly abstracted away by the Operating System; however, there are still some useful attributes that may aid in achieving portability, namely:

1. HARDSAME: The target hardware architecture is the same (or similar) to the original.
2. HAIMP: The target hardware architecture implements the same standards as the original architecture.

3.4 Computing the Environment Portability Index

The Environment Portability Index is computed in a similar fashion as the Software Portability Index. First, the software developer assigns values to as many attribute as possible in the Environment Portability Index Checklist based on the rating scale provided in 3.1.

Once the known attributes have been assigned a rating, the Environment Portability Index is computed by Equation 3.2 where E_{attrib} refers to each attribute in the Environment Portability Index Checklist and n refers to the total number of attributes in the Checklist. The value $S_{Environment}$ is a constant that represents the number of possible assignments for each attribute as listed in Table 3.1 and is multiplied by n to derive the maximum value of the Environment Portability Index Checklist which is used to normalize the Environment Portability Index to a value that ranges between 0 and 1.

$$EPI = \frac{\sum E_{attrib}}{S_{Environment} \sum n} \quad (3.2)$$

3.5 The Portability Index Threshold

In Chapter 1, brief mention was made of a user-defined Portability Index Threshold that would be useful for allowing a software developer to determine the lower bound of what value would be considered to be "good" portability in the context of the application under consideration. For example, if a developer specifies a Portability Index Threshold of 0.5, it means that any value that the Portability Indices take on that is greater than or equal to 0.5 is considered to be good portability.

In selecting a suitable Portability Index Threshold, developers must take into consideration the price they are willing to pay for including portability in the design of the software, the higher the price, the greater the Portability Index Threshold value. Since the Portability Indices will range between 0 and 1, the Portability Index Threshold must also range between 0 and 1.

3.6 Interpreting The Portability Indices

Once the developers have computed the Portability Indices and determined a reasonable Portability Index Threshold, the final step is to interpret the values to provide some meaningful information about the portability of the software being considered.

The average of the Portability Indices, $PI_{Average}$, is computed by Equation 3.3 and the value is compared to the Portability Index Threshold. The following are the possible interpretations of the values:

- If $PI_{Average} = 1$, then it may be possible to port with no modifications (perfect portability).
- If $PI_{Average} < Portability\ Index\ Threshold$, then redevelopment may be a preferable option over portability and the decision whether or not to attempt porting should be made by determining the disparity between the two values (which is a potential indicator of the degree of effort required to port the software).
- If $PI_{Average} \geq Portability\ Index\ Threshold$, then portability may be preferable to redevelopment and the greater the difference between the two values, the lower the difficulty of adapting the software to the new environment.

$$PI_{Average} = \frac{SPI + EPI}{2} \quad (3.3)$$

3.7 Weighted Attributes

Based on an extensive review of the literature, specifically [Moo97], [Cou02], [SUR96], [All02] and [TWN98] it is apparent that some design decisions are

more important to portability than others. In order to reflect this reality in the Portability Index, it is necessary to place weights on certain attributes to ensure that they contribute more to the estimation of the Portability and Effort Indices.

Value	Definition
1	Low
2	Medium
3	High
4	Very High

Table 3.2: Portability Indices Attribute Weighting Table

Table 3.2 lists the weights that are used for both the SPI and EPI Checklists. For simplicity, the weighting criteria are very similar to those used for both Checklists in table 3.1. The greater the contribution a particular attribute is perceived to have on portability, the larger its weight. Tables 3.3 and 3.4 provide an initial set of weights for each attribute in the SPI and EPI respectively. Note that the user-specified value for each attribute is multiplied by its corresponding weight to produce a new, weighted value.

Attribute	Weight
RED	2
PREQ	4
PLIM	2
PCOST	4
METH	4
ISOL	3
EXT	2
STAND	2
DISC	2
IMPSTAND	4
PLAN	1
PREC	2
WARN	2
SYSDEP	2
CONF	1
PDOC	3
VER	2
SIMP	3
LSTAND	2
LSUB	3
LENV	4
LUND	2
LMOD	4
OSSTAND	4
OSABS	4
OSPORT	4
OSOPEN	2
ARCSTAND	1

Table 3.3: Weights assigned to SPI Attributes

Attribute	Weight
OSSIM	4
OSIMP	4
LANGENV	3
LANGIMP	4
HARDSAME	3
HAIMP	3

Table 3.4: Weights assigned to EPI Attributes

The inclusion of weighted attributes requires that the equations for the Portability Indices described in 3.1 and 3.2 be re-written as equations 3.4 and 3.5, where Max_{score} is a constant that represents the maximum possible score that could be obtained if all the attributes were assigned the highest value in Table 3.1. W_{attrib} is used to represent the weight assigned to each attribute in the SPI and EPI computations. P_{attrib} and E_{attrib} represent the attributes for the SPI and the EPI respectively.

$$SPI = \frac{\sum W_{attrib} \times P_{attrib}}{Max_{score}} \quad (3.4)$$

$$EPI = \frac{\sum W_{attrib} \times E_{attrib}}{Max_{score}} \quad (3.5)$$

Note that Max_{score} is 300 for the SPI and 88 for the EPI using the current weights assigned to each attribute that comprise the indices.

3.8 Further Considerations

The initial Portability and Effort Index outlined in this chapter assumes that each attribute is equally weighted, which may not hold true in all cases. By applying weights to attributes based on how they are perceived to impact on portability, the author attempts to provide more meaningful and realistic values for the Portability Indices.

Though the concept of weighting the attributes is quite novel and appears to be in line with the idea of having the Portability Indices framework, it is highly debatable as to whether or not the weights specified truly represent the importance of each attribute to portability or even if the attributes themselves correctly capture all the considerations for portable design due to the fact that they were conceived based on interpretations of qualitative literature. Undoubtedly

a great deal of work is needed to formally establish the worth of the attributes and weights. However, the purpose of this thesis is to provide an initial structure for this framework and, as such, formal verification is beyond the scope of this work.

The next chapter will describe a series of experiments conducted on several hypothetical and real-world problems to illustrate how the metric could be used in different scenarios as well as providing some fundamental analysis to justify the worth of the measure.

Chapter 4

Portability Index Case Studies

This section will focus on two case studies that were conducted as a means of providing a preliminary degree of validation for the Portability Indices described in the previous chapter. The two case studies are based on the porting of a video game and a specialized compiler as described in [Moo93].

In order to conduct these case studies, a rudimentary tool was written in Perl that implemented the ideas discussed in Chapter 3. The full listing for this tool, and supporting documentation, is presented in Appendix A. The tool was designed to compute both weighted and unweighted versions of the Portability Indices.

For each case study, several experiments were conducted using the SPI and EPI to arrive at some conclusion regarding the portability of a particular design. In all cases, experiments were first conducted using the unweighted attributes and then repeated using the weighted ones, which were then compared to determine how similar the conclusions drawn in both cases were.

4.1 Case Studies

This section will highlight the details of the various case studies and provide discussions of the results gathered from applying the Portability Indices.

4.1.1 Case Study 1: Video Game

This case study involves a hypothetical video game system described in [Moo93]. An action-oriented video game that uses animated color graphics is being developed. The question here is to determine whether or not portability is to be incorporated into the design of the game. There are no specific details given about the class of architectures that this product will be ported to, except that the architecture must provide support for an effective color graphic interface, consistently smooth animation, as well as consistent timing.

Based on the description above, it may be inferred that the criteria could be met by many different PCs and workstations. [Moo93] lists the specific requirements for this type of program as:

1. PC or Workstation class.
2. Medium Resolution color graphics (e.g. 512 x 512, 256 colors).
3. Mouse or equivalent pointing device.

For the purpose of the discussion, it is assumed that an acceptable Portability Index Threshold is 0.5 because the interactive nature of the product demands high costs for interface functionality and optimal performance and too strict a portability requirement might have a negative impact on these factors. Furthermore, it will be assumed that C++ is the programming language used to implement the game and the product was originally designed to run on some Microsoft Windows-based system.

Experiment 1: No Portability Considerations In Software Lifecycle

In this experiment, it was assumed that portability was not considered at all as a part of the product's design, instead the developers chose to rely solely on the programming language and operating system to achieve portability. Table 4.1 shows the SPI Rating Scale (unweighted and weighted) used in the computation of the SPI.

Attribute	Score (Unweighted)	Score (Weighted)
RED	0	0
PREQ	0	0
PLIM	0	0
PCOST	0	0
METH	0	0
ISOL	0	0
EXT	0	0
STAND	0	0
DISC	0	0
IMPSTAND	0	0
PLAN	0	0
PREC	0	0
WARN	0	0
SYSDEP	0	0
CONF	0	0
PDOC	0	0
VER	0	0
SIMP	0	0
LSTAND	3	6
LSUB	3	9
LENV	3	12
LUND	4	8
LMOD	4	16
OSSTAND	2	8
OSABS	2	8
OSPORT	1	4
OSOPEN	0	0
ARCSTAND	1	1
SPI	0.21	0.24

Table 4.1: Video Game SPI Checklist (No Portable Design Considered)

Since no portability was considered in the original product, all attributes for

the software lifecycle were set to 0. The programming language being used was assumed to be C++, which has formally defined standards and provides a high degree of modularity, therefore it rated well in most categories. The Operating System category was fairly reasonable as well because Microsoft Windows can be considered a fairly popular operating system that uses some degree of standardization, internal to Microsoft. Finally, not much was known about the hardware that the software was being developed on, therefore the Hardware Architecture category scored relatively low in comparison to the others.

In both the unweighted and the weighted Portability Index calculations, the scores were very similar. In the unweighted instance, the score was 0.21 and in the case of the weighted index the score was 0.24.

The Environment Portability Checklist in Table 4.2 was computed based on the assumption that the developer was attempting to port the application to an environment that supported some Microsoft-based Operating System and had some version of the C++ compiler available for it.

Attribute	Score (Unweighted)	Score (Weighted)
OSSIM	4	16
OSIMP	2	8
LANGENV	3	9
LANGIMP	3	12
HARDSAME	0	0
HAIMP	0	0
EPI	0.50	0.55

Table 4.2: Video Game EPI Checklist

Since it is unsure what version of Microsoft Windows is running in the target environment, we assume that the operating system will be somewhat similar to Windows 98. In addition, there is a version of C++ available for the architecture, which implies that some portability gains can be made from there; however, it must be considered that the version of C++ on the target architecture may

pose portability problems based on how the compiler was implemented. Finally, nothing is known about the hardware that the system is running on, so no scores are assigned to the hardware architecture category. It is observed that the EPI estimated in both the unweighted and weighted cases only differed by a very small margin 0.05 with the unweighted value being 0.50 and the weighted value being 0.55.

Average (Unweighted)	Average (Weighted)
0.35	0.39

Table 4.3: Average of Video Game SPI and EPI (No Portable Design Considered)

With regard to the average of the Portability Indices, Table 4.3 shows the unweighted and weighted averages of 0.35 and 0.39 respectively. The margin of difference was very small between both values (0.04) and they both fell below the Portability Index Threshold of 0.5. In this case, the developers may determine that redevelopment is a better alternative to portability.

Experiment 2: Portability Consideration in Design

In this experiment we may assume that the developer now wishes to incorporate some degree of portability into the design of the video game. Table 4.4 illustrates the choices that the developer made to incorporate some portability with the purpose of striking a balance between the gains made by portable design and not making the software too portable so as to impact on its performance.

The EPI remains the same as in the previous experiment (Table 4.2). It can be seen that the SPI has improved drastically by incorporating portable design. The unweighted value of the SPI is 0.63 and the weighted value is 0.64.

Table 4.5 shows the unweighted and weighted averages have increased to of 0.56 and 0.59 respectively. In this case, the design decisions taken by the

developers have paid off by enabling them to get slightly more than they were willing to settle for based on the Portability Index Threshold.

Discussion of Results

In the first set of experiments, the weighted and unweighted Portability Indices were very similar and converged to the same general conclusion that portability was difficult to achieve when high-level design techniques in favor of portability were not considered. However, the second set of experiments showed that incorporating some degree of portable design into the software development resulted in a degree of portability that was considered suitable for the nature of application being considered.

The fact that the weighted and unweighted averages in the second set of experiments were very similar leads to some skepticism about the weighting scale being used, but more tests need to be conducted to determine how significant the weights are in this scenario.

4.1.2 Case Study 2: Special Purpose Compiler

This case study deals with the decision as to what degree of portable design should be incorporated into a compiler that will be applied to a very limited area. Unlike the video game in the previous case study, the compiler has a potential of being adapted to a wide range of vastly different architectures. Based on the fact that the compiler will only have limited application, it has been determined that the price that the developer is willing to pay for portability is not very high. Due to the nature of the compiler, we assume that the designers will use 0.40 as the Portability Index Threshold. In addition, we assume that Lex and Yacc are used to construct the compiler on an Intel PC architecture running Red Hat Linux.

Experiment 1: No Portability Considerations

In this experiment, it was assumed that portability was not considered at all as a part of the high-level design of the product. Table 4.6 shows the SPI Checklist (unweighted and weighted) used in the computation of the SPI.

In Table 4.6, no portability scores were given to any of the attributes for software lifecycle due to the fact that there was no portability consideration in the design. Under the category of language, Lex and Yacc are considered to be the most popular tools for constructing compilers; however, there are no known formal standards for these tools and it is unclear as to how modular they are in general. Red Hat Linux scored relatively highly as the Operating System category because it is known to be relatively consistent in implementing the POSIX interface standards; however there are still subtle differences that make it different from other flavors of Linux. A low hardware architecture score was assigned since the only thing known about the architecture is that it is based on the Intel PC platform, which may implement some degree of standardization, but it is unclear what specific architecture is used and what standards are incorporated into the design of the hardware.

It is observed that the weighted score for the SPI is just marginally better than the unweighted score, 0.21 for the unweighted compared to 0.25 for the weighted. However, both cases indicate that some portable design must be incorporated to achieve the threshold value of 0.4.

To compute the EPI, we assume that the only thing that is known about a potential target environment that the developer would like to port to is that it is based on the PowerPC architecture and runs an Operating System that implements the POSIX standard. As a result, the EPI Checklist shown in Table 4.7 only has a value assigned to the OSIMP attribute to reflect that the operating system in the new target environment, at the very least, implements

the POSIX standard. The unweighted instance of the EPI is approximately .01 points less than the weighted instance, so no marked difference can be found in their interpretation.

Table 4.8 shows the unweighted and weighted averages are 0.14 and 0.17 respectively. Both values lie below the Portability Index Threshold and, as such, very low portability gains may be anticipated from the product. The sizes of the averages are directly influenced by the fact that very little is known about the target environment being ported to.

Experiment 2: Portability Considerations in Design

In this experiment, the designer decided to choose scores for a few design attributes that are considered to be useful in gaining more portability in the design. Table 4.9 provides a listing of the attributes and the scores assigned to them.

From Table 4.9 it can be seen that the unweighted SPI value is not much lower than the weighted version.

Table 4.8 shows the unweighted and weighted averages are 0.25 and 0.27 respectively. Both values lie below the Portability Index Threshold and, as such, very low portability gains may be anticipated from the product.

At this point, the averages are still lower than the Portability Index Threshold and the developer may either opt to learn more about the target environment so as to get a better value for the Portability Indices, attempt to incorporate higher values for the attributes in the SPI Checklist, or consider redevelopment as a serious alternative.

Discussion of Results

In both experiments, the weighted and unweighted Portability Indices were very similar and converged to the same general conclusion that portability was not

achievable without high-level design to support a portable design. In the second set of experiments the developers selected SPI values that they thought would aid in portability without sacrificing too much in the way of the price they were willing to pay for the benefits. Unfortunately, the SPI computed by the choices yielded only a marginal improvement of the Portability Indices. A major part of the problem was that there was not sufficient information about the target environment to have an accurate measure of the EPI.

4.2 Merit of the Portability Indices

Based on the results seen from the case studies, it can be contended that the Portability Indices has the potential to be very useful measures for developers who wish to reap some quantifiable degree of benefit from portable design. It is interesting to note that, in the two examples presented, the unweighted and weighted scores were very similar and this may be an indicator that the weights used are not sufficient enough to represent the true importance of the various attributes used in the design or the less likely case that the weights really do not matter very much. Further research needs to be conducted with real data in order to establish better weighting criteria for the attributes.

Attribute	Score (Unweighted)	Score (Weighted)
RED	2	4
PREQ	4	16
PLIM	2	4
PCOST	2	8
METH	3	12
ISOL	3	9
EXT	3	6
STAND	1	2
DISC	3	6
IMPSTAND	2	8
PLAN	1	1
PREC	1	2
WARN	2	4
SYSDEP	4	8
CONF	4	4
PDOC	4	12
VER	3	6
SIMP	3	9
LSTAND	3	6
LSUB	3	9
LENV	3	12
LUND	4	8
LMOD	4	16
OSSTAND	2	8
OSABS	2	8
OSPORT	1	4
OSOPEN	0	0
ARCSTAND	1	1
SPI	0.63	0.64

Table 4.4: Video Game SPI Checklist (Portable Design Considered)

Average (Unweighted)	Average (Weighted)
0.56	0.59

Table 4.5: Average of Video Game SPI and EPI (Portable Design Considered)

Attribute	Score (Unweighted)	Score (Weighted)
RED	0	0
PREQ	0	0
PLIM	0	0
PCOST	0	0
METH	0	0
ISOL	0	0
EXT	0	0
STAND	0	0
DISC	0	0
IMPSTAND	0	0
PLAN	0	0
PREC	0	0
WARN	0	0
SYSDEP	0	0
CONF	0	0
PDOC	0	0
VER	0	0
SIMP	0	0
LSTAND	2	4
LSUB	0	0
LENV	2	8
LUND	3	6
LMOD	2	8
OSSTAND	3	12
OSABS	4	16
OSPORT	3	12
OSOPEN	4	8
ARCSTAND	1	1
SPI	0.21	0.25

Table 4.6: Specialized Compiler SPI Checklist(No Portable Design Considered)

Attribute	Score (Unweighted)	Score (Weighted)
OSSIM	0	0
OSIMP	2	8
OSDEPEND	0	0
LANGENV	0	0
LANGSIMP	0	0
HARDSAME	0	0
HAIMP	0	0
EPI	0.08	0.09

Table 4.7: Specialized Compiler EPI Checklist

Average (Unweighted)	Average (Weighted)
0.14	0.17

Table 4.8: Average of Specialized Compiler SPI and EPI (No Portable Design Considered)

Attribute	Score (Unweighted)	Score (Weighted)
RED	1	2
PREQ	1	4
PLIM	1	2
PCOST	1	4
METH	1	4
ISOL	2	6
EXT	1	2
STAND	1	2
DISC	2	4
IMPSTAND	3	12
PLAN	1	1
PREC	1	2
WARN	1	2
SYSDEP	2	4
CONF	1	1
PDOC	0	0
VER	1	2
SIMP	2	6
LSTAND	2	4
LSUB	0	0
LENV	2	8
LUND	3	6
LMOD	2	8
OSSTAND	3	12
OSABS	4	16
OSPORT	3	12
OSOPEN	4	8
ARCSTAND	1	1
SPI	0.41	0.45

Table 4.9: Specialized Compiler SPI Rating Scale (Portable Design Considered)

Average (Unweighted)	Average (Weighted)
0.25	0.27

Table 4.10: Average of Specialized Compiler SPI and EPI (Portable Design Considered)

Chapter 5

The Portable Code Generator

The Portable Code Generator is intended to be a highly flexible tool to aid in the design of portable software by enabling developers to analyze existing source code for potential portability problems and applying a series of transformations to the code to produce a more portable version in a language that may or may not be the same as the language the code was originally written in. The Portable Code Generator draws its inspiration from the concept of Grammatical Evolution [R⁺98] in that it uses an evolutionary approach to generate new code with very limited human interaction.

This chapter will provide an in-depth discussion of the architecture of the Portable Code Generator and Chapter 6 will provide the results of experiments conducted with a prototype version of the tool.

5.1 Architecture of the Portable Code Generator

The Portable Code Generator is comprised of four major components, namely:

1. The Code Analyzer
2. The Code Abstraction Unit
3. The Code Generator
4. The Code Tester

The Code Analyzer is responsible for evaluating source code to determine if there are any statements that could be a barrier to portability. The Code Abstraction Unit converts the original source code to a user-defined architecture independent form that is used by the Code Generator to generate new source code. Finally, the Code Tester is responsible for applying tests to the newly generated source code to determine if it produces the same results as the original code.

The design of the Portable Code Generator is intended to allow a high degree of flexibility and maintainability. As such, the four components of the tool are implemented in a highly modular fashion that makes it very simple for users to modify them or substitute their own custom modules without affecting the functionality of the software. Figure 5.1 illustrates a simple flow diagram of the PCG and will be used as the basis for describing the functionality offered by the tool.

The original source code is read into the Code Analyzer, which evaluates it for potential portability problems that may be caused by certain statements or programming practices. Once portability issues are identified and reported, the Code Abstraction Unit converts the code to a user-defined, machine independent format that represents the high-level functionality of the program, ignoring all implementation-specific details. The intermediate code produced by the Code Abstraction Unit is forwarded to the Code Generator, where it is transformed into new source code that may or may not be in the same language as the original source code. In the final stage, the Code Tester module executes a

Figure 5.1: Portable Code Generator Flowchart

variety of user-defined tests that will help to determine how well the code meets the desired requirements. If the code fails the tests, it will require some manual tuning of the machine-independent code to produce better results. This test-tune cycle will continue until the code meets the specified requirements. Once the code is determined to be acceptable, it is once more tested by the Code Analyzer to ensure that no additional portability issues have been incurred by the automatic code generation.

5.1.1 The Code Analyzer

The Code Analyzer is responsible for determining any possible portability issues that might arise due to non-standard uses of the programming language. This is not to say that the way the code is written is incorrect, it simply implies that there are certain syntactic or semantic conventions that may work for one platform and not another. For example, consider the following C++ code from [Wil98]:

```
FooBarClass static_object(87, 92);

void
bar()
{
    if (static_object.count > 15) {
        ...
    }
}
```

The code appears to be a harmless use of static constructors. However, [Wil98] observes that static constructors are usually instantiated just before the `main()` function is called at startup time and are usually comprised of two segments, a data segment that loads static data into the global data segment of the program and an initializer function that is called by the loader. The authors

of [Wil98] note that the initializer function is not implemented correctly in many C++ compilers and, as such, the object data is received but never initialized, which leads to portability problems depending on the compiler that is available for the platform being ported to.

[Wil98] suggests a portable solution to the use of static constructors in the following code segment:

```
FooBarClass*
getStaticObject()
{
    if (!static_object)
        static_object =
new FooBarClass(87, 92);
    return static_object;
}

void
bar()
{
    if (getStaticObject()->count > 15) {
        ...
    }
}
```

In this example, the non-portable code written in the first case is solved by including a wrapper function, *getStaticObject*.

Some programming languages, such as Ada, are inherently portable because the standards that have been established have been very strict and demand complete compliance with the standards in order to be considered an implementation of the language. Other languages, such as Fortran, are also standardized; however, the vagueness and incompleteness of the standards tend to be a prohibitive factor for portability of the language. In the case of languages that adhere strictly to standards, there is no real room for innovation in the language, which makes the language very resistant to changes.

Languages, such as C and C++, attempt to strike a balance between restrictive standards and vague standards. In many cases, these languages demonstrate very good portability across a variety of platforms; however, as the examples above illustrated, there are some very succinct caveats within these languages that can result in unforeseen barriers to portability.

Many languages have what is referred to as a "portable subset", which is simply a guide that illustrates what aspects of the language is implemented in a standardized fashion across a wide variety of architectures. The main source for the type of portability checking that will be carried out by the Code Analyzer will originate from these sources. Documents such as [Wil98] and [DLKR00] provide suitable guides (based on the extensive experiences of the authors) for implementing the type of rule checking required by the Code Analyzer.

5.1.2 The Code Abstraction Unit

The second major component of the Portable Code Generator is the Code Abstraction Unit. The idea of having a machine-independent intermediate representation for source code with the aim of converting it into a newer, more portable form is influenced directly by the work of [FF99] where the authors successfully implemented a tool, referred to as Bogart, that could convert code written in IBM 370 machine code to C code for use on newer architectures. The success of this approach as measured by Bogart's ability to convert several large modules comprising thousands of lines of code provides much inspiration for the approach taken by the Portable Code Generator.

Bogart analyzed the IBM 370 source code in order to determine the control and data flow. The result of the analysis was then stored in an abstract internal representation that was mapped to C code. The Code Abstraction Unit borrows heavily from the ideas used in the implementation of Bogart; however, the fact

that it is attempting to analyze code written in a high-level language, provides more challenges when compared to Bogart.

Due to the large number of high-level programming languages available and the fact that it is an extremely tedious task to establish a machine-independent language that is flexible enough to represent a wide range of programming languages, the representation used in the initial implementation of the Code Abstraction Unit will be implemented as a list of transformations that will map between the original source code and the new target source code. For example, consider the following statement written in the C language that simply prints "hello world *number*" 20 times:

```
for (i=0; i<20; i++) {  
    printf("hello world %d\n", i);  
}
```

The machine independent representation for the above code segment may be:

```
for-structure [i=0 i<20 i++]  
    display "hello world"  
    display i  
    new-line  
end for-structure
```

The machine-independent representation simply attempts to capture the logic of what the code segment is trying to accomplish. In the Code Generator, there will be a mapping that goes from the intermediate representation to the target language. For instance, if the language we wish to re-direct the source to was Perl, there would be a mapping that produces the following output:

```

for ($i=0; $i<20; i++) {
    print("hello world $i\n");
}

```

In [SUR96], the authors contend that logic-based languages, such as Prolog, are highly portable because they deal with the actual logic of what the program is supposed to achieve and leave the control flow up to the language itself. The Code Abstraction Unit is based around these ideas, where the intermediary language is intended to represent what the program is required to do (the logic) and not worry about how the program does it (the control flows).

5.1.3 The Code Generator

The Code Generator is the most important component of the Portable Code Generator. It is responsible for taking the high-level intermediate code and converting it into a more portable version that may be implemented in the same language as the original or a completely different one.

As the example in the previous section demonstrated, the Code Generator contains transformations from the intermediate language to the final target language. It is hoped that there is a perfect one-to-one mapping between each item in the intermediate code representation and the target language transformations; however, this is not often the case, because there are certain concepts that are unique to a particular language that may not be available in another. For example, C++ is an Object-Oriented language and Fortran is not, therefore it would be completely unrealistic to expect the Object-oriented concept of classes and inheritance to be easily implemented in a language that has no idea of these structures.

For the purpose of this research, it is assumed that all languages used in the studies will be relatively similar in nature. Future work will focus on producing a more versatile code generator capable of handling a wide range of programming

languages that are inherently different in nature, such as converting from Prolog to C and vice-versa.

5.1.4 The Code Tester

The final module to be discussed is the Code Tester. This module is intended to provide a measure of "fitness" for each program that is produced by the Code Generator. The Code Tester is intended to perform a similar function to the fitness measure used in Grammatical Evolution and other evolution-based algorithms [R⁺98]. Each time a program is generated by the Code Generator, the Code Tester will attempt to execute the module with a variety of test cases that determines how well it meets the requirements. The number of successful test cases determines the fitness of a particular module. The greater the number of successful test cases that a module accomplishes translates into a higher fitness value.

Once a module is determined to have a lower fitness than anticipated, the user must manually tune the machine-independent representation of the code and re-generate the code again. This cycle is repeated until the highest fitness level specified by the user is achieved, which means that the code accomplishes most (if not all) of its requirements. Later, users can apply their own manual modifications to the source code to resolve other issues that they may have with the automatically generated code.

For the purpose of simplicity, the Code Tester will not be implemented in the prototype. Instead, the testers will apply their own manual fitness tests to the automatically generated code.

5.2 Further Considerations

In this section we outlined the components of the Portable Code Generator and attempted to establish a reasonable scope of what the prototype of this tool is expected to accomplish. There are many other challenges associated with this type of tool that go beyond the scope of this research. It is anticipated that the work conducted here will provide a foundation for future portability work.

In the next chapter, we discuss the implementation of the Portable Code Generator and several case studies conducted to determine how reasonable the prototype model is.

Chapter 6

Applications of the Portable Code Generator

This chapter will discuss a rudimentary implementation of the Portable Code Generator (PCG) designed with the purpose of exploring the concept of applying evolutionary algorithms (specifically Grammatical Evolution [R⁺98]) and automatic programming to the problem of identifying potential portability problems in existing source code and being able to regenerate more portable versions of the code in a language that may or may not be the same language in which the original code was written.

In the previous chapter, the architecture of the PCG was discussed in detail. The version of the PCG presented here is in no way intended to be complete and is only capable of converting very rudimentary programs for analyzing and re-generating very small subsets of C++ code in three languages: namely Perl, C and C++. Further, only very simple versions of Code Analyzer, Code Abstraction Unit and Code Generator are implemented.

6.1 Implementation Language

Perl was chosen as the implementation language because of its very powerful pattern matching and file handling capabilities, to which the application owes the bulk of its success. The complete listing for the source code along with a concise users guide is presented in Appendix B.

6.2 Caveats

The PCG is in no way a completely automated code generator and still relies on a great deal of human input to complete tasks. Some of the major problems encountered with implementing the tool are:

- The program successfully identifies variables that are declared as integers, integer arrays, or characters but it has difficulty converting them to variables for other programming languages. Perl is extremely problematic in this regard because it does not do variable declarations but all variables are preceded by a \$ sign.
- The PCG is capable of identifying simple variable assignments, for example the C++ assignment statement *i=10;* poses no problem for the tool; however it runs in to difficulties with assignment operations such as *i=10*j;*.
- The current implementation of the PCG is capable of interpreting code with various functions, however it can only translate functions with return type declared as *int*.
- The PCG is very sensitive to programming style and, as a result, requires that things such as variable declarations, curly braces and so on must be on separate lines. For example:

```
int i,j;
```

must be written as:

```
int i; int j;
```

- The PCG is capable of interpreting C++ input and output statements (cin and cout), however it is incapable of handling multiple pairs of $\ll$ or $\gg$ in a single statement.

6.3 Training the Portable Code Generator

One of the major difficulties associated with automatic (or semi-automatic) code generation is the fact that there is no universal, machine independent code representation that is capable of mapping one language to any other language. In order to teach the PCG to identify various patterns in the C++ source code and establishing an intermediate language that could be mapped back to Perl, C and C++, the patterns of several example programs were manually coded into the Code Abstraction module along with pattern-matching rules on how to translate each line of the code into a high-level representation that was constantly refined to facilitate new features as they were encountered. The potential effect of this type of training is that, with the more examples the program sees, the more capable it will be of becoming more autonomous when translating code it has never encountered before.

C and Perl were deliberately chosen as target languages because of the fact that they are both very close in syntactical and semantic structure to C++, but sufficiently different to pose various challenges with regard to identifying suitable translations. In order to train the PCG, three programs were used. Figure 6.1 to Figure 6.3 illustrate the three C++ programs used to train the PCG.

```

#include <iostream>

int main()
{
    std::cout << "Hello World";

    return 0;
};

```

Figure 6.1: train1.cpp: Initial Training Program

The program illustrated in Figure 6.1 was used to teach the PCG how to identify header files, function declarations, basic text output and return values.

```

#include <iostream>

int main()
{
    int values[] = {1,2,3,4,5,6,7,8,9};
    int number;
    char value;

    for (int i = 0; i < 9; i ++)
    {
        number = rand();
        value = (char)rand();
        std::cout << "\nNumber:" << values[i];
        std::cout << "\nRandom:" << number;
        std::cout << "\nChar  :" << value;
    };

    return 0;
};

```

Figure 6.2: train2.cpp: Second Training Program

The program illustrated in Figure 6.2 was used to teach the PCG how to identify various variable types and handle for structures.

The program illustrated in Figure 6.3 was used to teach the PCG how to handle data input and multiple nested for statements.

```

#include <iostream>

int main()
{
    int inner;
    int outer;

    std::cout << "Enter an outer loop value: ";
    std::cin >> outer;
    std::cout << "Enter an inner loop value: ";
    std::cin >> inner;

    for (int o=0; o<outer; o++)
    {
        for (int i=0; i<inner; i++)
        {
            std::cout << "*";
        }
        std::cout << "\n";
    }
    return 0;
}

```

Figure 6.3: train3.cpp: Final Training Program

6.4 Resolution of C++ Portability Problems

In addition to learning how to match patterns to actual C++ statements, the Code Analyzer model was taught to identify a small subset of a larger set of portability-limiting coding practices as discussed in [Wil98]. The problems that the PCG learned to identify were:

1. **Put a new line at end-of-file.** Lack of a new-line character at the end of file causes problems with header files on the Sun WorkShop compiler and is also known to cause problems with .cpp files on HP. The PCG eliminates this problem by automatically adding a new line to the end of the file as a side effect of code translation.
2. **Do not put extra top-level semi-colons in code.** Putting a semi-

colon at the end of a statement block is known to cause problems with some compilers, such as the IBM AIX compiler. Code written as:

```
int
A::foo
{
};
```

would result in problems on some compilers and must be re-written as:

```
int
A::foo
{
}
```

The PCG automatically eliminates this problem by ignoring the semi-colon.

3. **Be careful of the scoping of variables declared inside for statements.** It is common practice to dynamically assign a new variable within a for loop, for example:

```
void
A::foo()
{
  for (int i = 0 ; i < 10 ; i++) {
    ...
```

would cause portability issues. The more portable alternative would be:

```
void
A::foo()
{
```

```
int i;
for (i = 0 ; i < 10 ; i++) {
...
}
```

The PCG is trained to identify these occurrences and capture them for incorporation into the regenerated version of the source code at the start of the function that houses the for statement in question.

6.5 Applying the Portable Code Generator

Once the Portable Code Generator was successfully able to translate the three initial programs into all three languages and eliminated any of the portability issues described above that had been intentionally placed in the programs the next step was to determine how well the PCG could translate several larger programs that it had never seen before given that it had learned how to translate a very rudimentary subset of the programming language. Three programs were tested. The first program, *factorial.cpp*, simply computed the factorial of a given number. The second program, *binary.cpp*, generated a list of random numbers and prompted the user to specify a number to search for and, if the number exists, return the position in the array that the number occupies. The final program to be tested was called *arrayedmo.cpp* and performed the same function as *binary.cpp*, but with a slightly different implementation. *factorial.cpp* was written by the author while the other two programs were downloaded from random web sites.

The results of the code translation are provided in Appendix C. It is important to note that none of the three programs translated completely correctly. One of the major issues that contributed to this was the fact that the training files could not sufficiently capture all the possible types of statements that the

tool would be exposed to. On the plus side, the code that was generated by the tool was, for the most part, readable and could be made to work with little effort and all the portability issues that the tool was designed to detect were filtered out.

6.6 Summary

This chapter discussed the implementation of a very rudimentary version of the Portable Code Generator that was capable of translating, with some degree of success, a small subset of various C++ statements to Perl and C. In addition, the tool was able to identify and eliminate a small subset of common C++ portability issues as outlined in [Wil98]. The code generated by the PCG though not perfect is understandable for the most part and requires relatively little effort to produce working code.

Chapter 7

Summary and Conclusion

7.1 Summary and Conclusion

The research conducted for this thesis resulted in the design of a potentially useful framework for providing developers with a means of selecting what techniques and considerations they wish to incorporate in their designs to support portability as well as establishing some meaningful value of the potential effort that will be required to port an application from the original environment it was developed for to a new environment.

The Software Portability Index is a useful tool for providing developers with a reasonable measure of the potential portability benefits that can be gained from incorporating various degrees of portability into their designs. It is intended to be a bargaining tool for managers and developers by allowing them to negotiate how much management is willing to pay for portability and determining how to maximize the benefits while minimizing the costs. The Software Portability Index is also used in conjunction with the Environment Portability Index to determine the probability of success of porting an existing piece of software to a newer environment. The success level is determined by the Portability Index Threshold, which is a user-defined measure that determines what "good"

portability is within the context of the software being developed and/or ported.

The major drawback of the Portability Indices described in this research is that the attributes and the weights used to compute them were based on the author’s interpretation of qualitative literature. Therefore, more work needs to be conducted to determine the viability of each attribute and its associated weight. However, the work provided here is undoubtedly a significant step in the right direction toward a good measure for portability.

The research also focused on the construction of a rudimentary tool known as the Portable Code Generator that was inspired by the evolutionary algorithms concept of Grammatical Evolution [O’N99b] to apply some degree of automation to the creation of newer, more portable source code from original code. The implementation of the Portable Code Generator described in this work was capable of translating very basic statements of C++ code to Perl and C. Although the code generated by the tool could not be compiled without the user altering it, the Portable Code Generator did successfully identify and eliminate the three categories of portability-hindering programming styles as described in [Wil98].

The concept of teaching the tool to handle various programs as it identifies them with the purpose of eliminating many portability issues is quite novel and requires further exploration.

7.2 Future Work

The following work may be considered for future project:

- Acquire and utilize real-world data that describes porting experiences to formally validate the Portability Indices metrics.
- Develop a more efficient version of the Portable Code Generator that is

capable of learning with minimal input from the user.

- Attempt to identify a better way of constructing the language-independent code.
- Incorporate support for more programming languages and portability issues.

Appendix A

Portability Index Estimation Tool

A.1 Overview

This appendix provides a brief overview of the tool used to implement and test the ideas discussed in Chapters 3 and 4. The tool was written as a Perl script because of the simplicity and flexibility of the language. The entire listing for the code is provided here as well as instructions for using it.

A.2 Usage Instructions

Inserting Attribute Values and Weights

In order to insert values for the SPI and EPI, open the script in your favorite text editor and modify the first few lines of the code. You may either use numbers (0-4) or the user-defined values of \$None, \$Low, \$Medium, \$High, \$VeryHigh. To same procedure is used to alter the weights for each attribute, except the weights are assigned values from 1-4 and have no mnemonic equivalent. Each attribute is listed using the general convention of \$ATTRIB and the weights for each attribute are listed in the source code as \$W_ATTRIB.

A.2.1 Execution

Once all the attribute and weights have been filled out, simply execute the program by issuing the command: *perl portindex.pl*. In a few moments, the weighted and unweighted values for the SPI and EPI will be listed.

A.3 Source Code Listing

```
#!/usr/bin/perl -w

# Portability Indices Estimation Tool v.0.0
# Author      : Gregory Mundy
# Date Created : May 19, 2004
# Last Modified : June 17, 2004
# Description  : This tool implements the ideas of the
#               Portability Indices described in Chapter 3
#               of this work.

####HIGH LEVEL PARAMETERS####
# Define values for the Rating Scale (Optional)

# $None      = 0;
# $Low       = 1;
# $Medium    = 2;
# $High      = 3;
# $VeryHigh  = 4;

# Define number of attributes in Portability Index
```

```

# Number of Attributes in SPI Checklist
$NumSPI    = 28;

# Number of Attributes in the EPI Checklist
$NumEPI    = 6;

# Maximum Score in Checklists
$MaxScore  = 4;

# Number of possible values in Rating Scale

# Maximum Weighted SPI Score
$MaxSPI_W  = 300;
# Maximum Weighted EPI Score
$MaxEPI_W  = 88;

# Define Portability Index Threshold
$Threshold = 0.5;

#### SPI Attributes ####

# Software Lifecycle

# Specifications
$RED       = 4;
$PREQ      = 4;
$PLIM      = 4;
$PCOST     = 4;

# Design
$METH      = 4;
$ISOL      = 4;
$EXT       = 4;
$STAND     = 4;

# Implementation
$DISC      = 4;
$IMPSTAND  = 4;

# Testing
$PLAN      = 4;
$PREC      = 4;
$WARN      = 4;

# Documentation
$SYSDEP    = 4;
$CONF      = 4;
$PDOC      = 4;

```

```

# Maintenance
$VER      = 4;
$SIMP     = 4;

# Language
$LSTAND   = 4;
$LSUB     = 4;
$LENV     = 4;
$LUND     = 4;
$LMOD     = 4;

# Operating System
$OSSTAND  = 4;
$OSABS    = 4;
$OSPORT   = 4;
$OSOPEN   = 4;

# Hardware Architecture
$ARCSTAND = 4;

####EPI Attributes####

# Operating System
$OSSIM    = 4;
$OSIMP    = 4;

# Language
$LANGENV  = 4;
$LANGIMP  = 4;

# Hardware Architecture
$HARDSAME = 4;
$HAIMP    = 4;

####SPI WEIGHTS####

# Software Lifecycle

# Specifications
$W_RED    = 2;
$W_PREQ   = 4;
$W_PLIM   = 2;
$W_PCOST  = 4;

# Design
$W_METH   = 4;
$W_ISOL   = 3;
$W_EXT    = 2;
$W_STAND  = 2;

```



```

# Implementation
$W_DISC      = 2;
$W_IMPSTAND  = 4;

# Testing
$W_PLAN      = 1;
$W_PREC      = 2;
$W_WARN      = 2;

# Documentation
$W_SYSDEP    = 2;
$W_CONF      = 1;
$W_PDOC      = 3;

# Maintenance
$W_VER       = 2;
$W_SIMP      = 3;

# Language
$W_LSTAND    = 2;
$W_LSUB      = 3;
$W_LENVP     = 4;
$W_LUND      = 2;
$W_LMOD      = 4;

# Operating System
$W_OSSTAND   = 4;
$W_OSABS     = 4;
$W_OSPORT    = 4;
$W_OSOPEN    = 2;

# Hardware Architecture
$W_ARCSTAND  = 1;

####EPI WEIGHTS####

# Operating System
$W_OSSIM     = 4;
$W_OSIMP     = 4;

# Language
$W_LANGENV   = 4;
$W_LANGIMP   = 4;

# Hardware Architecture
$W_HARDSAME  = 3;
$W_HAIMP     = 3;

####Compute Unweighted SPI####

```

```

# Lifecycle attributes
$SPI_U_Lifecycle = $RED + $PREQ + $PLIM + $PCOST + $METH + $ISOL + $
EXT
    + $STAND + $DISC + $IMPSTAND + $PLAN + $PREC + $WARN + $SYSDEP
    + $CONF + $PDOC + $VER + $SIMP;

# Language attributes
$SPI_U_Lang = $LSTAND + $LSUB + $LENV + $LUND + $LMOD;

# Operating System attributes
$SPI_U_Op = $OSSTAND + $OSABS + $OSPORT + $OSOPEN;

# Hardware attributes
$SPI_U_Hw = $ARCSTAND;

# Sum
$SPI_U = $SPI_U_Lifecycle + $SPI_U_Lang + $SPI_U_Op + $SPI_U_Hw;
#print (" $SPI_U\n");
$SPI_U = $SPI_U/($MaxScore * $NumSPI) ;

####Compute Unweighted EPI####

# Operating System
$EPI_U_Op = $OSSIM + $OSIMP;

# Language
$EPI_U_Lang = $LANGENV + $LANGIMP;

# Hardware
$EPI_U_Hw = $HARDSAME + $HAIMP;

# Sum
$EPI_U = $EPI_U_Op + $EPI_U_Lang + $EPI_U_Hw;
$EPI_U = $EPI_U/($MaxScore * $NumEPI);

####Compute Unweighted Average####
$AVG_U = ($SPI_U + $EPI_U)/2;

####Compute Weighted SPI####

# Lifecycle attributes
$SPI_W_Lifecycle = ($W_RED * $RED) + ($W_PREQ * $PREQ) + ($W_PLIM *
$PLIM)
    + ($W_PCost * $PCOST) + ($W_METH * $METH) + ($W_ISOL * $ISOL) +
($W_EXT * $EXT)
    + ($W_STAND * $STAND) + ($W_DISC * $DISC) + ($W_IMPSTAND * $IMPS
TAND)
    + ($W_PLAN * $PLAN) + ($W_PREC * $PREC) + ($W_WARN * $WARN) + ($

```

```

W_SYSDEP * $SYSDEP)
    + ($W_CONF * $CONF) + ($W_PDOC * $PDOC) + ($W_VER * $VER) + ($W_
SIMP * $SIMP);

# Language attributes
$SPI_W_Lang = ($W_LSTAND * $LSTAND) + ($W_LSUB * $LSUB) + ($W_LENv *
$LENv) + ($W_LUND * $LUND)
    + ($W_LMOD * $LMOD);

# Operating System attributes
$SPI_W_Op = ($W_OSSTAND * $OSSTAND) + ($W_OSABS * $OSABS) + ($W_OSPO
RT * $OSPORT)
    + ($W_OSOPEN * $OSOPEN);

# Hardware attributes
$SPI_W_Hw = ($W_ARCSTAND * $ARCSTAND);

# Sum
$SPI_W = $SPI_W_Lifecycle + $SPI_W_Lang + $SPI_W_Op + $SPI_W_Hw;
$SPI_W = $SPI_W/$MaxSPI_W;

####Compute Weighted EPI###

# Operating System
$EPI_W_Op = ($W_OSSIM * $OSSIM) + ($W_OSIMP * $OSIMP);

# Language
$EPI_W_Lang = ($W_LANGENV * $LANGENV) + ($W_LANGIMP * $LANGIMP);

# Hardware
$EPI_W_Hw = ($W_HARDSAME * $HARDSAME) + ($W_HAIMP * $HAIMP);

# Sum
$EPI_W = $EPI_W_Op + $EPI_W_Lang + $EPI_W_Hw;
$EPI_W = $EPI_W/$MaxEPI_W;

####Compute Weighted Average####
$AVG_W = ($SPI_W + $EPI_W)/2;

###Display Results###
print("\n\nPortability Index Threshold -> $Threshold\n\n");
print("SPI (Unweighted)           -> $SPI_U\n");
print("EPI (Unweighted)           -> $EPI_U\n");
print("AVG (Unweighted)             -> $AVG_U\n\n");
print("SPI (Weighted)               -> $SPI_W\n");
print("EPI (Weighted)               -> $EPI_W\n");
print("AVG (Weighted)               -> $AVG_W\n");

```

Appendix B

Portable Code Generator

B.1 Overview

This appendix provides a listing of the Portable Code Generator discussed in Chapters 5 and 6. Perl was chosen as the implementation language for the tool because of the flexibility of the language and its very powerful pattern matching abilities.

B.2 Usage Instructions

To utilize the PCG, issue the command: *perl pcg.pl c++ filename*. The argument *c++* indicates to the program that the file being analyzed is a C++ file and will be useful in the future to specify various types of programming languages. The *filename* argument represents path and name of a valid C++ file to analyze.

B.2.1 Files Created

The PCG creates several files that are used internally and externally by the program:

- **Analysis.txt** contains the portability problems detected by the Code Analyzer and is used by the Code Generator to ensure that the errors are caught.
- **Abstract.txt** contains the intermediary code generated by the Code Abstraction Unit and is used by the Code Generator to produce the new code.
- **Vars.txt** contains a list of variables detected for each function detected by the Code Abstraction Unit.
- **Output.pl** contains the re-generated code in Perl.
- **Output.c** contains the re-generated code in C.
- **Output.cpp** contains the re-generated code in C++.

B.3 Source Code Listing

```
#!/usr/bin/perl -w
```

```
#-----  
# Portable Code Generator |  
# Author      : Gregory Mundy |  
# Date Created : May 31, 2004 |  
# Last Modified : June 8, 2004 |  
# Description  : Analyze a module of code to determine |  
#               if any portability problems are |  
#               present. |
```

```

#-----

#-----
# Module Name: throwError |
# Purpose      : Error handler for the code generator. |
#               Displays an error message in response to |
#               an error and perform some user-defined |
#               action. |
# Inputs       : Error Code |
# Outputs      : Error message corresponding to error |
#-----

sub throwError {
    if ($_[0] == 000) {
        print("\nError000: Invalid number of command line arguments
specified!\n");
        exit(1);
    }

    if ($_[0] == 001) {
        print("\nError001: Unable to open file specified at command
line.\n");
        exit(1);
    }

    if ($_[0] == 002) {
        print("\nError002: Invalid language descriptor specified at
command line.\n");
        exit(1);
    }
}

#-----

# Module Name: loadLanguageModule |
# Purpose      : Loads modules for a language. |
# Inputs       : Descriptor of language module |
# Outputs      : None. |
#-----

sub loadLanguageModule {
    my $validLanguage = "FALSE"; # flag to ensure a valid language m
odule is specified.

    # load languages supported by this prototype.
    if ($_[0] eq "C" || $_[0] eq "c")
    {
        print("Loading C Language Module....\n");
        $validLanguage = "TRUE";
    }

    if ($_[0] eq "C++" || $_[0] eq "c++")

```

```

    {
        print("Loading C++ Language Module....\n");
        $validLanguage = "TRUE";
    }

    if ($_[0] =~/[Pp][Ll]/)
    {
        print("Loading Perl Language Module....\n");
        $validLanguage = "TRUE";
    }

    if ($_[0] =~/[Pp][Aa]$/)
    {
        print("Loading Pascal Language Module....\n");
        $validLanguage = "TRUE";
    }

    if ($validLanguage eq "FALSE") {
        throwError(002);
    }
}

#-----
# Module Name: checkArgs
# Purpose      : Checks the arguments specified at the
#               : command line to ensure validity.
# Inputs       : List of command line arguments.
# Outputs      : None.
#-----
sub checkArgs {

    # check number of arguments
    if ($numArgs != 2) {
        throwError(000);
    }

    # check if the file being analyzed exists
    open(,$ARGV[1]) || die(throwError(001));
}

sub parseFile {
    open(CODE, $ARGV[1]);

    do {
        $line = <CODE>;
        if ($line !~ /^s$/)
        {
            chomp($line);
            $line =~ s/^\s+|\s+$//g;
            push(@sourceCode, $line);
        }
    }
}

```

```

    }
} until eof (CODE);
return @sourceCode;
}

#-----
# Module Name: codeAnalyzer
# Purpose      : Analyzes the given code for potential
#                portability problems as a result of the
#                statements used to write the code.
# Inputs       : Original source code.
# Outputs      : Portability Log
#-----
sub codeAnalyzer{
    my $lineNumbers = $_[0] + 1;
    open(LOGFILE, ">$logFile");

    # Test 1: Don't put carriage returns in XP code
    # Note: As a side-effect of the stripping of whitespaces in
    # the previous section, this problem is reduced to a side effect

    # of the code.

    # Test 2: Don't put extra top level semi-colons in code
    for ($i=0;$i<$lineNumbers;$i++) {
        if($_[$i] =~ /\}\;/) {
            print($i);
            print(",");
            print($i);
            print(":");
            print("Extra top level semi-colon in code.\n");

            print LOGFILE $i;
            print LOGFILE ",";
            print LOGFILE $i;
            print LOGFILE ":";
            print LOGFILE "Rule 2\n";
        }
    }

    # Test 3: Be careful of variable declarations that require const
    ruction
    #                or initialization.
    for ($i=0; $i<$lineNumbers;$i++) {
        if ($_[$i] =~ /^[Ss][Ww][Ii][Tt][Cc][Hh]/ || $_[$i] =~ /^[Ii
    ][Ff]/) {
            $j = $i;

            while($_[$j] !~ /\}\;/) {
                $j++;
            }
        }
    }
}

```



```

        if ($_[ $j] =~ /^[A-Za-z]+\s[A-Za-z]+\([A-Za-z0-9]+\)/)
    /){
        print($i);
        print(",");
        print($j);
        print(":");
        print("Variable declaration that requires constr
uction/initialization.\n");

        print LOGFILE $i;
        print LOGFILE ",";
        print LOGFILE $j;
        print LOGFILE ":";
        print LOGFILE "Rule 3\n";
    }
}

}

# Test 4: Be careful of the scoping of variables declared inside
for() statements.

for($i=0;$i<$lineNumbers;$i++) {
    if ($_[ $i] =~ /^[Ff][Oo][Rr][\s]*\([A-Za-z]*[\s]*[A-Za-z]+[\s]*[A-Za-z0-9]+[\s]*=[\s]*[0-9]+[\s]*\;/ ) {
        print($i);
        print(",");
        print($i);
        print(":");
        print("Scoping of variables declared inside for statemen
ts.\n");

        print LOGFILE $i;
        print LOGFILE ",";
        print LOGFILE $i;
        print LOGFILE ":";
        print LOGFILE "Rule 4\n";
    }
}

# Test 5: Declare local initialized aggregates as static
for($i=0;$i<$lineNumbers;$i++) {
    if ($_[ $i] =~ /^[A-Za-z]+\s*[\s]*[A-Za-z0-9]+\s*\[\]\s*\s*\s*=[\s]*\{[\s]*\s*[A-Za-z0-9]+\s*\}/) {
        print($i);
        print(",");
        print($i);
        print(":");
        print("Declare local initialized aggregates as static.\n
");
    }
}

```

```

        print LOGFILE $i;
        print LOGFILE ",";
        print LOGFILE $i;
        print LOGFILE ":";
        print LOGFILE "Rule 5\n";
    }
}

# Test 6: Put a new line at the end of file
$x = $_[$lineNumbers - 1];
if($x ne "\n") {
    print($lineNumbers);
    print(",");
    print($lineNumbers);
    print(":");
    print("Put new line at the end of file.\n");

    print LOGFILE $i;
    print LOGFILE ",";
    print LOGFILE $i;
    print LOGFILE ":";
    print LOGFILE "Rule 6\n";
}
close(LOGFILE);
}

#-----
# Module Name: codeAbstraction |
# Purpose      : Convert source code from a specific |
#               programming language format to a machine |
#               and language independent format. |
# Inputs       : Original source code, Portability Log |
# Outputs      : Machine-Independent Code Representation |
#-----
sub codeAbstraction {
    my $lineNumbers = $#_ + 1;
    $forStructure = "null";
    $numFor = 0;

    open(ABSTRACT, ">$abstractFile");
    open(VARS, ">$varsFile");

    for ($i=0;$i<$lineNumbers;$i++)
    {

        # Abstraction Rules Learned from Basic1.cpp
        if ($_[$i] =~ /^#\s*[Ii][Nn][Cc][Ll][Uu][Dd][Ee]/) {
            $includeLib = $_[$i];

```

```

$includeLib =~ s/#include[\s]*/LIBRARY /gi;
$includeLib =~ s/\<//;
$includeLib =~ s/\>//;
print ABSTRACT (" $includeLib\n");
}

if ($_[ $i] =~ /[Ii][Nn][Tt]\s*[A-Za-z0-9_]+\s*(\s*[A-Za-z0-9;,_\s\[\]]*\s*\))/) {
    $j=$i;
    $leftParen = 0;
    $rightParen = 0;
    $exitFlag = -1;

    $subRoutine = $_[ $i];
    $subRoutine =~ s/int\s*/SUB INTEGER /gi;
    $subRoutine =~ s/\(\)//gi;
    print ABSTRACT (" $subRoutine\n");
    print VARS (" $subRoutine\n"); # write the variables for
the subroutine to file

    do {
        $j++;

        if ($_[ $j] =~ /\{/ ) {
            $leftParen++;
        }

        if ($_[ $j] =~ /\}/ ) {
            $rightParen++;
        }

        if ($_[ $j] =~ /[Ss][Tt][Dd]\:\:[Cc][Oo][Uu][Tt]/) {
            $output = $_[ $j];
            $output =~ s/std\:\:cout\s*/DISPLAY/gi;
            $output =~ s/\<\>//gi;
            $output =~ s/\; //gi;
            print ABSTRACT (" $output\n");
        }

        # Abstraction rules learned from Basic3.cpp
        if ($_[ $j] =~ /[Ss][Tt][Dd]\:\:[Cc][Ii][Nn]/) {
            $input = $_[ $j];
            $input =~ s/std\:\:cin\s*/ACCEPT/gi;
            $input =~ s/\>\>//gi;
            $input =~ s/\; //gi;
            print ABSTRACT (" $input\n");
        }
    }
}

```

```

if ($_[ $j] =~ /[Rr][Ee][Tt][Uu][Rr][Nn]/) {
    $return = $_[ $j];
    $return =~ s/return/RETURN/gi;
    $return =~ s/;/;/;

    print ABSTRACT (" $return\n");
}

if ($leftParen == $rightParen && $rightParen ne 0) {

    print ABSTRACT ("END SUB\n");
    $exitFlag = 0;
}

# Abstraction rules learned from Basic2.cpp

# Translate an assignment operation

if ($_[ $j] =~ /\s*[A-Za-z0-9_]+\s*=\s*[A-Za-z0-9_+\-\\/%\(\)]+\s*/) {
    if ($_[ $j] !~ /[Ff][Oo][Rr]/) {
        $assign = $_[ $j];
        $assign =~ s/;/;/;
        $assign =~ s/^ \s*(.*) \s*$/$1/gi;
        print ABSTRACT (" $assign\n");
    }
}

# Translate a basic Integer Declaration

if ($_[ $j] =~ /[Ii][Nn][Tt]\s*[A-Za-z0-9_]+\s*;/) {

    $varName = $_[ $j];
    $varName =~ s/^ \s*(.*) \s*$/$1/gi;
    $varName =~ s/;/;/;
    $varName =~ s/int/INTEGER/gi;
    print VARS (" $varName\n");
}

# Translate Integer Arrays

if ($_[ $j] =~ /[Ii][Nn][Tt]\s*[A-Za-z0-9_]+\s*\[\s*[0-9]*\s*\]/) {
    if ($_[ $j] =~ /[Ii][Nn][Tt]\s*[A-Za-z0-9_]+\s*\[\s*[0-9]*\s*\]\s*=\s*\{\s*[0-9,]+\s*\}/) {
        $varName = $_[ $j];
        $varData = $_[ $j];
        $arrayLen = $_[ $j];
    }
}

```

```

$varName =~ s/int//gi;
$varName =~ s/=//;
$varName =~ s/\\{\s*[0-9,]+\s*\\}//gi;
$varName =~ s/;/;/;
$varName =~ s/\\[\s*[0-9]*\s*\\]\\//gi;
$varName =~ s/^\\s*(.*?)\\s*$/1/gi;

$varData =~ s/int//gi;
$varData =~ s/=//;
$varData =~ s/[A-Za-z0-9_]+//i;
$varData =~ s/;/;/;
$varData =~ s/\\[\s*[0-9]*\s*\\]\\//gi;
$varData =~ s/\\[//;
$varData =~ s/\\]//;
$varData =~ s/^\\s*(.*?)\\s*$/1/gi;

$arrayLen =~ s/int//gi;
$arrayLen =~ s/=//;
$arrayLen =~ s/\\{\s*[A-Za-z0-9_,]+\s*\\}\\//gi;

$arrayLen =~ s/[A-Za-z0-9_]+//i;
$arrayLen =~ s/;/;/;
$arrayLen =~ s/\\[//gi;
$arrayLen =~ s/\\]//gi;
$arrayLen =~ s/^\\s*(.*?)\\s*$/1/gi;

if ($arrayLen !~ /[0-9]+/) {
    $arrayLen = NULL;
}

print VARS ("INTEGER ");
print VARS ("$varName ");
print VARS ("$arrayLen ");
print VARS ("$varData\\n");
}
else {
    $varName = $_[$j];
    $arrayLen = $_[$j];

    $varName =~ s/int//gi;
    $varName =~ s/\\{\s*[0-9,]+\s*\\}\\//gi;
    $varName =~ s/;/;/;
    $varName =~ s/\\[\s*[0-9]*\s*\\]\\//gi;
    $varName =~ s/^\\s*(.*?)\\s*$/1/gi;

    $arrayLen =~ s/int//gi;
    $arrayLen =~ s/\\{\s*[A-Za-z0-9_,]+\s*\\}\\//gi;

```

```

$arrayLen =~ s/[A-Za-z0-9_]++/i;
$arrayLen =~ s/;/;
$arrayLen =~ s/\[//;
$arrayLen =~ s/\]//;
$arrayLen =~ s/^\s*(.*?)\s*$/1/gi;

if ($arrayLen !~ /[0-9]+/) {
    $arrayLen = NULL;
}

print VARS ("INTEGER ");
print VARS ("$varName ");
print VARS (" $arrayLen\n");
}
}

# Translate a basic Character Declaration
if ($_[ $j] =~ /[Cc][Hh][Aa][Rr]\*\s*[A-Za-z0-9_]+\s
*\;/) {

    $varName = $_[ $j];
    $varName =~ s/^\s*(.*?)\s*$/1/gi;
    $varName =~ s/\;/;
    $varName =~ s/char/CHARACTER/gi;
    print VARS (" $varName\n");
}

# Translate a for loop
if ($_[ $j] =~ /[Ff][Oo][Rr]/) {
    if ($_[ $j] =~ /[Ff][Oo][Rr]\s*\(\s*[Ii][Nn][Tt]\s
s*[A-Za-z0-9_]+\s*\=/) {
        $numFor++;
        $loopValue = $_[ $j];
        $loopValue =~ s/for//gi;
        $loopValue =~ s/int/INTEGER/gi;
        $loopValue =~ s/\s*=\s*[0-9]+\s*//;
        $loopValue =~ s/;\s*[A-Za-z0-9+-<=>\s]+\s*;\s
s*[A-Za-z0-9+-<=>\s]++//gi;
        $loopValue =~ s/=//;
        $loopValue =~ s/\(//;
        $loopValue =~ s/\)//;
        $loopValue =~ s/^\s*(.*?)\s*$/1/gi;
        print VARS (" $loopValue\n");

        $forStructure = $_[ $j];
        $forStructure =~ s/for/FOR-STRUCTURE/gi;
        $forStructure =~ s/int//gi;
        $forStructure =~ s/\(//;
        $forStructure =~ s/\)//;
        $forStructure =~ s/^\s*(.*?)\s*$/1/gi;

```

```

        print ABSTRACT ("$forStructure\n");
    }
    else {
        $numFor++;

        $forStructure = $_[$j];
        $forStructure =~ s/for/FOR-STRUCTURE /gi;
        # $forStructure =~ s/\(//;
        # $forStructure =~ s/\)//;
        $forStructure =~ s/~\s*(.*?)\s*$/1/gi;

        print ABSTRACT ("$forStructure\n");
    }
}

if ($_[$j] =~ /\}/ && $numFor > 0) {
    print ABSTRACT ("END FOR-STRUCTURE\n");
    $numFor--;
}
} while ($exitFlag == -1);

$i = $j;
}
}
close(ABSTRACT);
close(VARS);
}

#-----
# Module Name: codeGenerator |
# Purpose      : Generates code based on the high level |
#               representation created by the codeAbstr- |
#               action module. |
# Inputs       : Machine-Independent Code Representation |
# Outputs      : Portable Source Code |
#-----
sub codeGenerator {

    open(MODEL, "abstract.txt");
    open(VARS, "vars.txt");
    open(CFILE, ">output.c");
    open(PERLFILE, ">output.pl");
    open(CPPFILE, ">output.cpp");

```

```

do {
    $abstract=<MODEL>;
    push(@abstractCode,$abstract);
} until eof(MODEL);
close(MODEL);

do {
    $var=<VARS>;
    push(@variables,$var);
} until eof(VARS);
close(VARS);

# Translation rules learned from Basic1.cpp

for ($i=0;$i<$#abstractCode+1;$i++) {
    $iUnderstand = -1;

    if ($abstractCode[$i] =~ /LIBRARY/) {
        $iUnderstand = 0;
        $library = $abstractCode[$i];
        $library =~ s/library//gi;

        if ($library =~ /[Ii][Oo][Ss][Tt][Rr][Ee][Aa][Mm]/) {
            print CFILE ("#include <stdio.h>\n");
            print PERLFILE ("#!/usr/bin/perl -w\n");
            print CPPFILE ("#include <iostream>\n");
        }
    }

    if ($abstractCode[$i] =~ /SUB\sINTEGER/) {
        $iUnderstand = 0;
        $subName = $abstractCode[$i];

        $subName =~ s/sub\s*integer//gi;
        $subName =~ s/^\s*(.*?)\s*$/1/;

        print CFILE ("int $subName()\n{\n");
        print PERLFILE ("sub $subName {\n");
        print CPPFILE ("int $subName()\n{\n");
    }

    if ($abstractCode[$i] =~ /DISPLAY/) {
        $iUnderstand = 0;
        $displayText = $abstractCode[$i];
        $displayText =~ s/DISPLAY//;
    }
}

```



```

$displayText =~ s/^\s*(.*?)\s*$/1/;

print CFILE ("printf(\"%s\",$displayText);\n");
print PERLFILE ("print $displayText;\n");
print CPPFILE ("std::cout << $displayText;\n");
}

if ($abstractCode[$i] =~ /RETURN\s[0-9]+/) {
    $iUnderstand = 0;
    $return = $abstractCode[$i];
    $return =~ s/RETURN/return/gi;
    $return =~ s/^\s*(.*?)\s*$/1/;

    print CFILE ("return;\n");
    print PERLFILE ("return;\n");
    print CPPFILE ("return;\n");
}

if ($abstractCode[$i] =~ /END/) {
    $iUnderstand = 0;
    print CFILE ("}\n");
    print PERLFILE ("}\n");
    print CPPFILE ("}\n");
}

# Translation Rules Learned from Basic2.cpp

if ($abstractCode[$i] =~ /^FOR-STRUCTURE/) {
    $iUnderstand = 0;
    $forStmtC = $abstractCode[$i];
    $forStmtPl = $abstractCode[$i];
    $forStmtC =~ s/FOR-STRUCTURE/for/gi;
    $forStmtC =~ s/^\s*(.*?)\s*$/1/;

    $forStmtPl =~ s/FOR-STRUCTURE/for/gi;
    $forStmtPl =~ s/([A-Za-z0-9_]*\s*[=<>+-]+\s*)/\$1/gi;
    $forStmtPl =~ s/(;\s*[A-Za-z0-9_]\$]+\s*[=\<>\+|-]+)(\s*[A-Za-z_]+\s*)(\;)/$1\$2\$3/gi;
    $forStmtPl =~ s/^\s*(.*?)\s*$/1/;

    print CFILE ("{$forStmtC {\n");
    print PERLFILE ("{$forStmtPl {\n");
    print CPPFILE ("{$forStmtC {\n");
}

# Translation Rules Learned from Basic3.cpp
if ($abstractCode[$i] =~ /ACCEPT/) {
    $iUnderstand = 0;
    $getText = $abstractCode[$i];

```

```

$getText =~ s/ABSTRACT//;
$getText =~ s/^\s*(.*?)\s*$/1/;

$getC = $getText;
$getCxx = $getText;
$getCpl = $getText;

$getC = s/ACCEPT/scanf/;
$getCxx = s/ACCEPT/std\:\:cin/gi;
$getCpl = s/ACCEPT//gi;

$getC = s/(scanf\s*)([A-Za-z0-9_]+)/$1\("%d",\&$2\
\;/;

$getCxx = s/(std\:\:cin\s*)([A-Za-z0-9_]+)/$1\>\>$2\;/;
$getCpl = s/([A-Za-z0-9_]+)/$1\<STDIN\>\;chomp($\
$1)\;/;

$getCpl = s/^\s*(.*?)\s*$/1/;

print CFILE ("getC\n");
print PERLFILE ("getCpl\n");
print CPPFILE ("getCxx\n");
}

if ($abstractCode[$i] =~ /[A-Za-z0-9_\[\]\s]+\s*=\s*[A-Za-z
0-9_\+\-\*\(\)]+\s*/) {
    $iUnderstand = 1;

    $abstractCode[$i] = s/^\s*(.*?)\s*$/1/;

    print CFILE ("$abstractCode[$i]\;\n");
    print PERLFILE ("$abstractCode[$i]\;\n");
    print CPPFILE ("$abstractCode[$i]\;\n");
}

if ($iUnderstand == -1) {
    print ("I do not understand $abstractCode[$i]");
}
}

print PERLFILE ("main;");
close(CFILE);
close(PERLFILE);
close(CPPFILE);
}

#-----
#                               Main Program Section                               |
#-----
#system("clear");                # clear screen
$logFile = "analysis.txt";      # log file name

```

```

$abstractFile = "abstract.txt"; # abstract file
$varsFile = "vars.txt";         # variable storage file
$numArgs = $#ARGV + 1;          # number of arguments passed into co
mmmand line

checkArgs;                       # check validity of arguments
loadLanguageModule($ARGV[0]);    # loads the appropriate source langu
age module
@code = parseFile;               # parse the source code specified at
the command line
codeAnalyzer(@code);             # run the code analyzer over the exi
sting code
codeAbstraction(@code);          # run the code abstraction unit
codeGenerator;                   # run the code generator

```

Appendix C

Results from Portable Code Generator

C.1 train1.cpp

C.1.1 Original C++ Code

```
#include <iostream>

int main()
{
    std::cout << "Hello World";

    return 0;
};
```

C.1.2 Abstract Code

```
LIBRARY iostream
SUB INTEGER main
DISPLAY "Hello World"
RETURN 0
END SUB
```

C.1.3 Re-generated C Code

```
#include <stdio.h>
int main()
{
    printf("%s","Hello World");
    return 0;
}
```

C.1.4 Re-generated Perl Code

```
#!/usr/bin/perl -w
sub main {
    print "Hello World";
    return 0;
}
main;
```

C.1.5 Re-generated C++ Code

```
#include <iostream>
int main()
{
    std::cout << "Hello World";
    return 0;
}
```

C.2 train2.cpp

C.2.1 Original C++ Code

```
#include <iostream>

int main()
{
    int values[] = {1,2,3,4,5,6,7,8,9};
    int number;
    char value;

    for (int i = 0; i < 9; i ++)
    {
        number = rand();
        value = (char)rand();
        std::cout << "\nNumber:" << values[i];
        std::cout << "\nRandom:" << number;
        std::cout << "\nChar  :" << value;
    };

    return 0;
};
```

C.2.2 Abstract Code

```
LIBRARY iostream
SUB INTEGER main
FOR-STRUCTURE ( i = 0; i < 9; i ++)
number = rand()
value = (char)rand()
DISPLAY "\nNumber:" values[i]
DISPLAY "\nRandom:" number
DISPLAY "\nChar  :" value
END FOR-STRUCTURE
RETURN 0
END SUB
```

C.2.3 Re-generated C Code

```
#include <stdio.h>
int main()
{
    for ( i = 0; i < 9; i ++) {
        number = rand();
        value = (char)rand();
        printf("%s","\nNumber:" values[i]);
        printf("%s","\nRandom:" number);
        printf("%s","\nChar  :" value);
    }
}
```

```

    return 0;
}

```

C.2.4 Re-generated Perl Code

```

#!/usr/bin/perl -w
sub main {
    for ( $i = 0; $i < 9; $i ++ ) {
        number = rand();
        value = (char)rand();
        print "\nNumber:" values[i];
        print "\nRandom:" number;
        print "\nChar  :" value;
    }
    return 0;
}
main;

```

C.2.5 Re-generated C++ Code

```

#include <iostream>
int main()
{
    for ( i = 0; i < 9; i ++ ) {
        number = rand();
        value = (char)rand();
        std::cout << "\nNumber:" values[i];
        std::cout << "\nRandom:" number;
        std::cout << "\nChar  :" value;
    }
    return 0;
}

```

C.3 train3.cpp

C.3.1 Original C++ Code

```

#include <iostream>

int main()
{
    int inner;
    int outer;

    std::cout << "Enter an outer loop value: ";
    std::cin >> outer;
    std::cout << "Enter an inner loop value: ";
    std::cin >> inner;
}

```

```

    for (int o=0; o<outer; o++)
    {
        for (int i=0; i<inner; i++)
    {
        std::cout << "*";
    }
        std::cout << "\n";
    }
    return 0;
}

```

C.3.2 Abstract Code

```

LIBRARY iostream
SUB INTEGER main
DISPLAY "Enter an outer loop value: "
ACCEPT outer
DISPLAY "Enter an inner loop value: "
ACCEPT inner
FOR-STRUCTURE ( o=0; o<outer; o++)
FOR-STRUCTURE ( i=0; i<inner; i++)
DISPLAY "*"
END FOR-STRUCTURE
DISPLAY "\n"
END FOR-STRUCTURE
RETURN 0
END SUB

```

C.3.3 Re-generated C Code

```

#include <stdio.h>
int main()
{
    printf("%s","Enter an outer loop value: ");
    $1("%d",&$2);
    printf("%s","Enter an inner loop value: ");
    $1("%d",&$2);
    for ( o=0; o<outer; o++) {
        for ( i=0; i<inner; i++) {
            printf("%s","*");
        }
        printf("%s","\n");
    }
    return 0;
}

```

C.3.4 Re-generated Perl Code

```

#!/usr/bin/perl -w

```



```

sub main {
    print "Enter an outer loop value: ";
    outer
    print "Enter an inner loop value: ";
    inner
    for ( $o=0; $o<$outer; $o++) {
        for ( $i=0; $i<$inner; $i++) {
            print "*";
        }
        print "\n";
    }
    return 0;
}
main;

```

C.3.5 Re-generated C++ Code

```

#include <iostream>
int main()
{
    std::cout << "Enter an outer loop value: ";
    std::cin >> outer;
    std::cout << "Enter an inner loop value: ";
    std::cin >> inner;
    for ( o=0; o<outer; o++) {
        for ( i=0; i<inner; i++) {
            std::cout << "*";
        }
        std::cout << "\n";
    }
    return 0;
}

```

C.4 factorial.cpp

C.4.1 Original C++ Code

```

#include <iostream>

int main()
{
    int fact;
    int fact1;

    std::cout << "Please enter the number you would like to compute the factorial of:";
    std::cin >> fact;

    fact1=fact;

```

```

    for (int i=1; i<fact; i++)
    {
        fact1 = fact1 * i;
        std::cout << fact1;
        std::cout << "\n";
    }

    std::cout << "The factorial is " << fact1;

    return 0;
}

```

C.4.2 Abstract Code

```

LIBRARY iostream
SUB INTEGER main
DISPLAY "Please enter the number you would like to compute the factorial of:"
ACCEPT fact
fact1=fact
FOR-STRUCTURE ( i=1; i<fact; i++)
fact1 = fact1 * i
DISPLAY fact1
DISPLAY "\n"
END FOR-STRUCTURE
DISPLAY "The factorial is " fact1
RETURN 0
END SUB

```

C.4.3 Re-generated C Code

```

#include <stdio.h>
int main()
{
    printf("%s","Please enter the number you would like to compute the factorial of:");
    $1("%d",&$2);
    fact1=fact;
    for ( i=1; i<fact; i++) {
        fact1 = fact1 * i;
        printf("%s",fact1);
        printf("%s","\n");
    }
    printf("%s","The factorial is " fact1);
    return 0;
}

```

C.4.4 Re-generated Perl Code

```

#!/usr/bin/perl -w
sub main {

```

```

        print "Please enter the number you would like to compute the factorial of:";
        fact
fact1=fact;
    for ( $i=1; $i<$fact; $i++) {
fact1 = fact1 * i;
print fact1;
print "\n";
    }
    print "The factorial is " fact1;
    return 0;
}
main;

```

C.4.5 Re-generated C++ Code

```

#include <iostream>
int main()
{
    std::cout << "Please enter the number you would like to compute the factorial of:";
    std::cin >>fact;
    fact1=fact;
    for ( i=1; i<fact; i++) {
        fact1 = fact1 * i;
        std::cout << fact1;
        std::cout << "\n";
    }
    std::cout << "The factorial is " fact1;
    return 0;
}

```

C.5 arraydemo.cpp

C.5.1 Original C++ Code

```

#include <iostream>

int inputa(int a[ ]);
int outputa(int a[ ]);
int dosort(int a[ ]);
int swap(int &x, int &y);
int search(int a[ ]);
int dosearch(int a[ ], int b);

int main()
{
    int a[10];
    inputa(a);
    outputa(a);

```

```

        std::cout<<"\n";
        dosort(a);
        outputa(a);
search(a);
return 0;
}
int inputa(int a[ ])
{
    int i;
    for(i=0; i<10;i++)
    {
        a[i]=rand()%101;
    }
}
int outputa(int a[ ])
{
    int pig;
    for(pig=0; pig<10;pig++)
    {
        std::cout<<a[pig]<<"\n";
    }
}
int dosort(int a[ ])
{
    int j;
    int pig;
    for(pig=0;pig<10;pig++)
    {
        for(j=0;j<10;j++)
        {
            if (a[j]>a[pig])
            {
                swap(a[j],a[pig]);
            }
        }
    }
}
int swap(int &a, int &b)
{
    int temp;
    temp=a;
    a=b;
    b=temp;
}
int search(int a[ ])
{
    int b;
    std::cout<<"Enter Number to find: ";
    std::cin>>b;
    dosearch(a,b);
}

```

```

    std::cout<<"The number was found in line number "<<dosearch(a,b)<<"\n";
}
int dosearch(int a[ ], int b)
{
    int high=10;
    int mid=5;
    int low=-1;
    int d=0;
    int line=0;
    do
    {
        if(b>a[mid])
    {
        low=mid+1;
        mid=(high+low)/2;
    }
        else if (b<a[mid])
    {
        high=mid+1;
        mid=(high+low)/2;
    }
    else if(a[mid]==b)
    {
        return mid+1;
    }
    }
    while(low<high);
    return -1;
}

```

C.5.2 Abstract Code

```

LIBRARY iostream
SUB INTEGER inputa(SUB INTEGER a[ ]);
DISPLAY"\n"
RETURN 0
END SUB
SUB INTEGER inputa(SUB INTEGER a[ ])
FOR-STRUCTURE (i=0; i<10;i++)
END FOR-STRUCTURE
END SUB
SUB INTEGER outputa(SUB INTEGER a[ ])
FOR-STRUCTURE (pig=0; pig<10;pig++)
DISPLAYa[pig]"\n"
END FOR-STRUCTURE
END SUB
SUB INTEGER dosort(SUB INTEGER a[ ])
FOR-STRUCTURE (pig=0;pig<10;pig++)
FOR-STRUCTURE (j=0;j<10;j++)
END FOR-STRUCTURE

```

```

END FOR-STRUCTURE
END SUB
SUB INTEGER search(SUB INTEGER a[ ])
DISPLAY"Enter Number to find: "
ACCEPTb
DISPLAY"The number was found in line number "dosearch(a,b)"\n"
END SUB
SUB INTEGER dosearch(SUB INTEGER a[ ], SUB INTEGER b)
int high=10
int mid=5
int low=-1
int d=0
int line=0
low=mid+1
mid=(high+low)/2
high=mid+1
mid=(high+low)/2
RETURN mid+1
RETURN -1
END SUB

```

C.5.3 Re-generated C Code

```

#include <stdio.h>
int inputa( a[ ] );()
{
    printf("%s","\n");
    return 0;
}
int inputa( a[ ] )()
{
    for (i=0; i<10;i++) {
    }
}
int outputa( a[ ] )()
{
    for (pig=0; pig<10;pig++) {
printf("%s",a[pig]"\n");
    }
}
int dosort( a[ ] )()
{
    for (pig=0;pig<10;pig++) {
        for (j=0;j<10;j++) {
        }
    }
}
int search( a[ ] )()
{
    printf("%s","Enter Number to find: ");
}

```

```

    $1("%d",&$2);
    printf("%s","The number was found in line number "dosearch(a,b)"\n");
}
int dosearch( a[ ], b)()
{
    int high=10;
    int mid=5;
    int low=-1;
    int d=0;
    int line=0;
    low=mid+1;
    mid=(high+low)/2;
    high=mid+1;
    mid=(high+low)/2;
}

```

C.5.4 Re-generated Perl Code

```

#!/usr/bin/perl -w
sub inputa( a[ ] ); {
    print "\n";
    return 0;
}
sub inputa( a[ ] ) {
    for ($i=0; $i<10;$i++) {
    }
}
sub outputa( a[ ] ) {
    for ($pig=0; $pig<10;$pig++) {
print a[pig]" \n";
    }
}
sub dosort( a[ ] ) {
    for ($pig=0;$pig<10;$pig++) {
for ($j=0;$j<10;$j++) {
    }
    }
}
sub search( a[ ] ) {
    print "Enter Number to find: ";
    $$1=<STDIN>;chomp($$1);
    print "The number was found in line number "dosearch(a,b)"\n";
}
sub dosearch( a[ ], b ) {
    int high=10;
    int mid=5;
    int low=-1;
    int d=0;
    int line=0;
    low=mid+1;

```

```

        mid=(high+low)/2;
        high=mid+1;
        mid=(high+low)/2;
    }
    main;

```

C.5.5 Re-generated C++ Code

```

#include <iostream>
{
    std::cout << "\n";
    return 0;
}
int inputa( a[ ] )()
{
    for (i=0; i<10;i++) {
    }
}
int outputa( a[ ] )()
{
    for (pig=0; pig<10;pig++) {
        std::cout << a[pig]"\n";
    }
}
int dosort( a[ ] )()
{
    for (pig=0;pig<10;pig++) {
    for (j=0;j<10;j++) {
    }
    }
}
int search( a[ ] )()
{
    std::cout << "Enter Number to find: ";
    std::cin>>b;
    std::cout << "The number was found in line number "dosearch(a,b)"\n";
}
int dosearch( a[ ], b )()
{
    int high=10;
    int mid=5;
    int low=-1;
    int d=0;
    int line=0;
    low=mid+1;
    mid=(high+low)/2;
    high=mid+1;
    mid=(high+low)/2;
}

```


C.6 binary.cpp

C.6.1 Original C++ Code

```
#include <iostream>

// Generate numbers (from 1-100)
// Print number of occurrences of each number
int main()
{
    const int DIFFERENT_NUMBERS = 100;

    // Prompt user to enter and read number of games.
    int totalNumbers;
    std::cout << "How many numbers to generate?: ";
    std::cin >> totalNumbers;

    int numbers[DIFFERENT_NUMBERS + 1];

    // Initialize the vector to zeros.
    for(int i=0; i < DIFFERENT_NUMBERS + 1; i++ )
    {
        numbers[i] = 0;
    }

    // Generate the numbers.
    for( int j = 0; j < totalNumbers; j++ )
    {
        numbers[ rand( ) % DIFFERENT_NUMBERS + 1 ]++;
    }

    // Output the summary.
    for( int k = 1; k <= DIFFERENT_NUMBERS; k++ )
    {
        std::cout << k << " occurs " << numbers[ k ]
        << " time(s)\n";
    }

    return 0;
}
```

C.6.2 Abstract Code

```
LIBRARY iostream
SUB INTEGER main
const int DIFFERENT_NUMBERS = 100
DISPLAY "How many numbers to generate?: "
ACCEPT totalNumbers
FOR-STRUCTURE( i=0; i < DIFFERENT_NUMBERS + 1; i++ )
END FOR-STRUCTURE
```

```

FOR-STRUCTURE( j = 0; j < totalNumbers; j++ )
END FOR-STRUCTURE
FOR-STRUCTURE( k = 1; k <= DIFFERENT_NUMBERS; k++ )
DISPLAY k " occurs " numbers[ k ]
END FOR-STRUCTURE
RETURN 0
END SUB

```

C.6.3 Re-generated C Code

```

#include <stdio.h>
int main()
{
    const int DIFFERENT_NUMBERS = 100;
    printf("%s", "How many numbers to generate?: ");
    $1("%d", &$2);
    for( i=0; i < DIFFERENT_NUMBERS + 1; i++ ) {
    }
    for( j = 0; j < totalNumbers; j++ ) {
    }
    for( k = 1; k <= DIFFERENT_NUMBERS; k++ ) {
        printf("%s", k " occurs " numbers[ k ]);
    }
    return 0;
}

```

C.6.4 Re-generated Perl Code

```

#!/usr/bin/perl -w
sub main {
    const int DIFFERENT_NUMBERS = 100;
    print "How many numbers to generate?: ";
    totalNumbers
for( $i=0; $i < $DIFFERENT_NUMBERS + 1; $i++ ) {
}
    for( $j = 0; $j <$ totalNumbers; $j++ ) {
    }
    for( $k = 1; $k <=$ DIFFERENT_NUMBERS; $k++ ) {
print k " occurs " numbers[ k ];
    }
    return 0;
}
main;

```

C.6.5 Re-generated C++ Code

```

#include <iostream>
int main()
{

```

```

const int DIFFERENT_NUMBERS = 100;
std::cout << "How many numbers to generate?: ";
std::cin >> totalNumbers;
for( i=0; i < DIFFERENT_NUMBERS + 1; i++ ) {
}
for( j = 0; j < totalNumbers; j++ ) {
}
for( k = 1; k <= DIFFERENT_NUMBERS; k++ ) {
    std::cout << k " occurs " numbers[ k ];
}
return 0;
}

```

Bibliography

- [AKS00] Erik R. Altman, David Kaeli, and Yaron Sheffer. Welcome to the Opportunities of Binary Translation. *IEEE Computer*, March 2000.
- [All02] Chuck Allison. Is Portability Still Important? *C/C++ Users Journal*, 2002.
- [B⁺95] Christian Bischof et al. A Case Study of MPI: Portable and Efficient Libraries. *In Proceedings of the seventh Conference on Parallel Processing for Scientific Computing*, 1995.
- [Bel98] Alan Bell. Software Portability. *Tessella Support Services PLC Technical Supplement*, 1998.
- [BN00] Greg J. Badros and David Notkin. A Framework for Preprocessor-Aware C Source Code Analyses. *Software Practice and Experience*, 30(8):907–924, 2000.
- [Coh83] Danny Cohen. a Plea for Peace. *IEEE Micro*, August 1983.
- [Cor01] Progress Software Corporation. Progress Portability Guide, May 2001.
- [Cou02] Alexandre Courbot. Porting the GNU Compiler to the Camille Architecture. Master’s thesis, Universite des Sciences et Technologies de Lille, 2002.
- [CR93] Richard Waters Charles Rich. Approaches to Automatic Programming. *Advances in Computers*, 37, 1993.
- [CS98] Kenneth Cain and Brian Sroka. Portable Software Library Optimization, 1998.
- [Dem00] T. Demir. Genetic algorithms, 2000.
- [DLKR00] A. Dolenc, A. Lemmke, D. Keppel, and G.V. Reilly. Notes on Writing Portable Programs in C, 2000.
- [Eig96] Rudolf Eigenmann. Portable parallel programming languages. *In Proceedings of International Conference on Parallel Processing*, 1996.

- [FF99] Yishai A. Feldman and Doron A. Friedman. Portability by Automatic Translation A Large-Scale Case Study. *Artificial Intelligence*, 107(1):1–28, 1999.
- [Fra94] Michael Steffen Oliver Franz. *Code-Generation On-the-Fly: A Key to Portable Software*. PhD thesis, Swiss Federal Institute of Technology Zurich, 1994.
- [Fra97] Michael Franz. Beyond Java: An Infrastructure for High-Performance Mobile Code on the World Wide Web. *In Proceedings of WebNet 97*, pages 33–38, October 1997.
- [FV03] D. Faust and C Verhoef. Software product line migration and deployment. *Software Practice and Experience*, 33(10):933–955, 2003.
- [Ger01] Rainer Gerlich. Platform-Dependent Cost Impacts on Portability, Software Reuse and Maintenance. *In Proceedings of Data Systems in Aerospace 2001*, 2001.
- [Gre93] Timothy G. Greenwood. International Cultural Differences in Software. *Digital Technical Journal*, 5(3), 1993.
- [H⁺98] M. Gonzalez Harbour et al. PINROB: A Portable API for Industrial Robots. *In Proceedings of ADA-Europe 98*, 1998.
- [Han03] David R. Hanson. Lcc.NET: Targeting the .NET Common Intermediate Language from Standard C, 2003.
- [Har92] David Harel. Biting the Silver Bullet Toward a Brighter Future for System Development. *IEEE Computer*, 25(1):8–20, 1992.
- [Hay03] B. Hayes. The Post OOP Paradigm. *American Scientist*, 9(2):106–110, 2003.
- [How95] James Howatt. A Project-Based Approach to Programming Language evaluation. *ACM SIGPLAN Notices*, 30(7):37–40, 1995.
- [K⁺99] J. Koza et al. Genetic Programming as a Darwinian Invention Machine. *In Proceedings Second European Workshop on Genetic Programming (EUROGP99)*, 1999.
- [KM02] Ren-Song Ko and Matt W. Mutka. FRAME for Achieving Performance Portability within Heterogeneous Environments. *In Proceedings of the 9th IEEE Conference on Engineering Computer Based Systems*, 2002.
- [Koz90] John R. Koza. Genetically Breeding Populations of Computer Programs to Solve Problems in Artificial Intelligence. *In Proceedings of the second International Conference on Tools for AI*, pages 819–827, 1990.

- [Koz95] J. Koza. Survey of Genetic Algorithms and Genetic Programming. *In Proceedings WesCon95*, 1995.
- [Koz97] J. Koza. Genetic programming, 1997.
- [L⁺96] C.H. Levy et al. IUP/LED: A Portable User Interface Development Tool. *Software Practice and Experience*, 26(7):737–762, 1996.
- [Lee02] Hyo-Kyung Lee. A Theory of Portability. *In Proceedings of International Conference of Language Resources and Evaluation*, 2002.
- [Lew02] Bruce Lewis. Software Portability Gains Realized with Metah and Ada95. *In Proceedings of the eleventh International Workshop on Real Time Ada*, 2002.
- [ML90] M.A. Murray-Lasso. Cultural and Social Constraints on Portability. *Journal of Research on Computing in Education*, 23(2), 1990.
- [Moo65] George Moore. Moore’s law. *Popular Electronics*, 1965.
- [Moo90] James D. Mooney. Strategies for Supporting Application Portability. *IEEE Computer*, 23(11):59–70, November 1990.
- [Moo93] James D. Mooney. Issues in the Specification and Measurement of Software Portability. *Technical Report TR 93-6 West Virginia University*, 1993.
- [Moo95] James D. Mooney. Portability and Reusability: Common Issues and Differences. *In Proceedings of twenty-third ACM conference on Computer Science*, pages 150–156, 1995.
- [Moo97] James D. Mooney. Bringing Portability to the Software Process. *Technical Report TR-97-1 West Virginia University*, 1997.
- [Moo99] James D. Mooney. Strategies for adaptation. *Draft*, 1999.
- [Moo02] Jim Moore. The Meaning of Portability. *Technical Support*, August 2002.
- [O⁺01] Michael O’Neill et al. Developing a Market Timing System using Grammatical Evolution. *In Proceedings of the Genetic and Evolutionary Computation Workshop Program*, 2001.
- [OMI95] OMI. Omi/spam. <http://www.cordis.lu/esprit/src/projlist.htm>, 1995.
- [O’N99a] Michael O’Neill. Automatic Generation of programs with Grammatical Evolution. *In Proceedings of the Genetic and Evolutionary Computation Workshop Program*, 1999.

- [O’N99b] Michael O’Neill. Automatic Programming with Grammatical Evolution. *In Proceedings of the Genetic and Evolutionary Computation Workshop Program*, 1999.
- [OR99] Michael O’Neill and Conor Ryan. Under the Hood of Grammatical Evolution. *In Proceedings of the Genetic and Evolution Computation Conference*, 1999.
- [Pou94] Jeffrey S. Poulin. Measuring Software Reusability. *In Proceedings of Third International Conference on Software Reuse*, 1994.
- [R⁺98] Conor Ryan et al. Grammatical Evolution: Evolving Programs for an Arbitrary Language. *In Proceedings of the first European Workshop on Genetic Programming*, 1998.
- [Sel94] John Sellens. SPP - Low Tech, Practical, UNIX Software Portability. *In Proceedings USENIX Application Development Symposium*, 1994.
- [SG03] James Snell and Tom Glover. Portability and Interoperability Similarities and Differences Explained. *IBM DeveloperWorks Web Services*, 2003.
- [SJ97] Horia C. Slusanschi and Chris R. Jesshope. The benefits of using the F-code Portable Software Platform. *In Proceedings of the fourth annual Australasian Conference on Parallel and Real-Time Systems*, 1997.
- [SUR96] Giancarlo Succi, Carl Uhrik, and Marco Ronchetti. Reusability and Portability of logic programming. *Journal of Programming Languages*, 4:101–114, 1996.
- [TWN98] Kenji Takeda, Ivan Wolton, and Denis Nicole. Software Portability and Maintenance. *In Proceedings of High Performance Computing Initiative 1998*, 1998.
- [Ull91] Ellen Ullman. You Can’t Run on Everything. *BYTE Magazine*, pages 255–264, November 1991.
- [Wil98] David Williams. C++ Portability Guide, 1998.
- [WR95] Paul Walsh and Conor Ryan. Automatic conversion of programs from serial to parallel using Genetic Programming - The Paragen System. *In Proceedings ParCo 95*, 1995.
- [ZT00] Cindy Zheng and Carol Thompson. PA-RISC to IA-64: Transparent Execution, No Recompilation. *IEEE Computer*, March 2000.