ANALYSIS AND IMPLEMENTATION OF
SOFTWARE REUSE MEASUREMENT

by

Carol G. Terry

Project and Report submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

MASTERS OF INFORMATION SYSTEMS

APPROVED:

W. Frakes, Chairman

C. Egyhazy

C. Fox

October, 1993

Blacksburg, Virginia
Analysis and Implementation of
Software Reuse Measurement

by Carol G. Terry

ABSTRACT

Software reuse has been shown to increase quality and productivity [Card et al 86] [Browne et al 90] [Frakes 91] [Agresti and Evanco 92]. As researchers and development organizations begin to recognize the potential benefits of systematic reuse of software, formal measures of the amount of reuse in a given system or subsystem are needed. A formal measurement of software reuse will provide software developers and managers with the necessary data to track reuse progress. This project and report describe such a measurement of parts-based reuse, building upon the reuse level metric and the rl software tool as described by Frakes in [Frakes 90] and [Frakes 92].

This paper reviews the current research literature in the areas of software reuse and software reuse metrics. The reuse metrics proposed by Frakes are extended to include reuse frequency and a reuse complexity weighting. The metrics are formally defined. Results from extensive testing of rl are reported and correlated with program size. The enhancements made to the rl program include:

- specification of the reuse frequency metric,
- an additional call graph abstraction for reuse measurement,
- weighting of software components for complexity,
- allowing the user to specify the number of uses of a software element which indicate reuse,
- and providing multiple choices for abstraction entities.
ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Bill Frakes, for his significant contribution to this project and report. He provided many ideas along with constant encouragement that pushed me to meet my goals. His professional expertise in the field of software reuse is very evident and I consider myself fortunate to have had the opportunity to work with him.

I would also like to acknowledge the support from my husband Ed that persisted throughout the project. His patience for tolerating the demands of the project during a "long" summer was admirable. I could not have made it without his love and forgiveness.

Thanks to all my friends, family, and colleagues who were forced to listen to the daily details of academia as I completed my Master's degree. Thanks for the kind ears and encouraging words. I would like to especially acknowledge my parents and siblings who taught me the value of education and the rewards of persistence and ambition.
TABLE OF CONTENTS

1 Introduction ......................................................................................... 1
  1.1 Problem Definition ................................................................. 1
  1.2 Organization of this paper ....................................................... 2
2 Literature Review ............................................................................. 4
  2.1 Formal Software Metrics ......................................................... 4
    2.1.1 Software complexity metrics ......................................... 5
    2.1.2 Objective and algorithmic measurements ..................... 5
    2.1.3 Process and product metrics ......................................... 6
    2.1.4 Models of the software development process ............... 6
    2.1.5 Meta-metrics ................................................................. 7
  2.1.6 Summary ............................................................................ 7
  2.2 Software Reuse .......................................................................... 8
    2.2.1 Software Reuse Terminology ......................................... 9
    2.2.2 The Emerging Technology of Software Reuse ............. 11
    2.2.3 Software Reuse in the Object-Oriented Environment ...... 13
    2.2.4 Implementation of Software Reuse ............................... 18
  2.3 Software Reuse Measurement .................................................... 21
    2.3.1 Existing Software Reuse Metrics .................................... 22
      2.3.1.1 Economic Models ................................................ 24
      2.3.1.2 Maturity Models ............................................... 29
      2.3.1.3 Reuse Ratio Models ......................................... 32
      2.3.1.4 Reuse Measurement in the Object-Oriented
        Environment ......................................................... 35
    2.3.2 Measuring Software Reuse Potential ............................. 37
    2.3.3 Relation of Reuse to Quality and Productivity ............... 38
  2.4 Literature Review Summary ....................................................... 42
3 The Reuse Metrics ........................................................................... 44
  3.1 Introduction .............................................................................. 44
LIST OF TABLES

Table 1: Types of Software Reuse ........................................................................................................... 10
Table 2: Software Reuse in C vs. C++ ....................................................................................................... 17
Table 3: Questions and Answers about Software Reuse Implementation ........................................ 19
Table 4: Models for Reuse Measurement .................................................................................................. 23
Table 5: Barnes' and Bollinger's economic investment model ............................................................... 27
Table 6: Costs and payoff threshold values for reusable components ............................................... 28
Table 7: Hudson and Kolton Reuse Maturity Model .................................................................................. 31
Table 8: Characteristics of SEL subsystems [Agresti and Evanco 92] ......................................................... 39
Table 9: Mean Development Time and 95% Confidence Intervals in Hours ....................................... 40
Table 10: Mean Number of Errors and 95% Confidence Intervals ....................................................... 40
Table 11: Reuse level values for Figure 1 ................................................................................................. 48
Table 12: Reuse level values for the multical graph abstraction shown in Figure 2 ................................. 50
Table 13: Reuse frequency values for Figure 1. ....................................................................................... 53
Table 14: Reuse frequency values for the multical graph abstraction shown in Figure 2. ......................... 53
Table 15: Reuse Metric / Abstraction Matrix .............................................................................................. 59
Table 16: Summary Statistics for Reuse Level .......................................................................................... 72
Table 17: Summary Statistics for Reuse Frequency .................................................................................. 74
LIST OF FIGURES

Figure 1: Call graph abstraction for higher level item H ........................................... 47
Figure 2: Multicall graph abstraction for higher level item H .................................... 49
Figure 3: rl Reuse Frequency Output ........................................................................ 61
Figure 4: rl Reuse Complexity Weighting Output .................................................... 62
Figure 5: Sample rl output ....................................................................................... 66
Figure 6: Internal Reuse Level vs. log NCSL ............................................................ 73
Figure 7: External Reuse Level vs. log NCSL ............................................................ 74
Figure 8: Internal reuse level vs. Internal threshold level ......................................... 75
Figure 9: External reuse level vs. External threshold level ....................................... 76
1 **Introduction**

1.1 **Problem Definition**

Many computer professionals believe that software is the most expensive component of a system. Hardware and communication capabilities have improved to a level that exceeds the speed and efficiency with which software can be developed. [Boehm 81] With this problem in mind, corporate managers are actively seeking progressive and effective methods to improve the software development process [Keyes 92]. An important part of improving the development process is the ability to track progress and measure the amount of improvement that occurs over time [Conc et al 86].

Software reuse is recognized as a method for dramatically reducing the time and expense of software development. By using existing software, development time is reduced and software quality is improved [Card et al 86]. As organizations implement software reuse programs in an effort to improve productivity and reliability, they must be able to measure their progress and identify what reuse strategies are most effective. To satisfy the need to measure reuse, this project and report present an empirical approach to reuse measurement. The objective of this work is to extend the *reuse level* [Frakes 90] metric, define the metric formally, and demonstrate the metric in an enhanced version of the *rl* [Frakes 92] program.
1.2 Organization of this paper

This paper presents a summary of software metrics and an in-depth review of software reuse metrics. The proposed extensions to and applications of the reuse level metric are explained in detail.

Chapter One gives a global definition of the problem and states the purpose of this project and report.

Chapter Two provides important background information in the areas of software measurement and software reuse. Justification for a software reuse measurement is given along with a thorough review of the literature in this area.

The actual metrics proposed by this research are presented in Chapter Three. An informal overview will acquaint the reader with the terms and definitions necessary for more detailed explanations. The reuse level and reuse frequency metrics are discussed in detail, along with a formal description of the metrics.

Chapter Four discusses the rI program which has been developed to measure software reuse in C programs. A detailed description of the software is given as well as samples of its use.
Chapter Five contains a discussion of testing the rl software. Conclusions on the amount of reuse found in the set of test programs are given, as well as correlations between the amount of reuse and program size.

Finally, Chapter Six presents a discussion of future work and conclusions obtained from this research.

The appendices contain the source code for the rl software and relevant documentation. They also contain listings of the source code used to test the rl program.
2 Literature Review

2.1 Formal Software Metrics

As in all engineering disciplines, measurement of software products and processes provides a quantitative analysis which can be used to improve the engineering process. Sommerville defines a software metric as "any measurement which relates to a software system, process or related documentation." [Sommerville 89]. It is important to define precise metrics so that different applications of a metric to the same program will obtain identical results. The field of software measurement has received abundant attention in recent literature; researchers are striving to reject the common criticism of poor empirical methodology and lack of theoretical foundation within the field [Baker et al 90].

In their book Software Engineering Metrics and Models, Conte, Dunsmore and Shen [Conte et al 86] identify five broad attributes of software metrics: software complexity metrics, objective and algorithmic measurements, process and product metrics, models of the software development process, and meta-metrics. As background information for future discussions, the next five sections explain each of these attributes. Much of this information is from [Conte et al 86].
2.1.1 Software complexity metrics

The complexity of a unit of software is determined by characteristics of the software itself as well as its interaction with other systems. A complexity metric reflects the difficulty one encounters in the design, coding, testing, or maintaining of the software system. Types of complexity include problem complexity, design complexity, and product complexity. Valid complexity metrics can be obtained objectively and have direct impact on program development metrics such as effort. Common complexity measures include McCabe's cyclomatic number, which uses graph theoretic techniques, and Halstead's programming effort, which measures complexity by considering the number of unique operators and operands and their frequency in a program [Sommerville 89]. A very simple measure of complexity is a count of the number of source lines of code in a program [Nejmeh 88]. As a program increases in size, it also increases in complexity. An equivalent size measure includes consideration for reused code vs. newly developed code [Conte et al 86].

2.1.2 Objective and algorithmic measurements

According to Conte, et. al., an objective, or algorithmic, measurement "is one that can be computed precisely according to an algorithm. Its value does not change due to changes in time, place, or observer." This definition implies the mathematical reliability of an objective measurement, allowing comparable results for research. Objective measurements have historically been difficult; researchers often disagree on how to define a metric, and for some abstract concepts, there are no algorithms that accurately capture them.
2.1.3 Process and product metrics

Software metrics are applied to the software product or to the software development process. Product metrics are measures of the software product. Products are deliverables, artifacts, or, in general, documents that result from the activities during the development life cycle. Example product metrics are size of the program, logic structure complexity, and data structure complexity. Process metrics quantify attributes of the development process and of the development environment. Resource metrics are process metrics, measuring the experience of personnel or the cost of development. Effort and cost measurements are extremely important to management of development projects. They are divided into two categories: the micro-level of measurement for effort expended by individual programmers on small projects, and the macro-level of measurement for effort expended by teams of programmers on large projects.

2.1.4 Models of the software development process

Software development models are mathematical models that deal with the software development process. A model is represented by the general form

\[ y = f(x_1, x_2, \ldots, x_n) \]

where \( y \) is the dependent variable and \( x_1, x_2, \ldots, x_n \) are independent variables. In a software development model, the dependent variable is a product or process metric. The independent variables are product- or process-related. For example, the dependent
variable might be development cost or effort, and the independent variables might include product complexity and the amount of reuse in the product.

A model may be theoretical or data-driven. Theoretical models are based on hypothesized relationships among factors, independent of actual data. Data-driven models are the result of statistical analysis of data obtained in empirical testing.

2.1.5 Meta-metrics

A meta-metric is a measure of a software metric. Conte, et. al. suggest that simplicity, reliability, validity, robustness, prescriptiveness, and analyzability are properties, or metrics, which can be used to evaluate a proposed metric. Other sources [Prather 84][Fenton and Melton 90][Weyuker 88] propose a formal framework in which complexity measures can be compared and contrasted. Specific axioms are presented which may be applied to a metric to assess its validity and reliability.

2.1.6 Summary

Many different types of metrics exist to measure aspects of software throughout the development cycle. Design metrics attempt to measure program modularization as well as the amount of coupling, cohesion and complexity within the design. Defect metrics determine the amount of errors or defects within a program; three typical metrics are: 1) number of changes required in the design, 2) number of errors, and 3) number of program changes. Software reliability metrics are related to defects within a system, providing the
probability of no failure during a given time interval. Metrics are also defined to assess the quality and completeness of a testing strategy.

The recent trend of software development organizations toward total quality management has increased the importance of quality measurement. Developers are recognizing the value of quality products and quality processes to achieve those products [Keyes 92]. Inherent in the push for quality is the need for measurement so that progress can be tracked and improved. A reliable and complete measurement technique is imperative for any engineering discipline. As Conte, et. al. state, "Systematic collection of [...] useful metrics is a necessary prerequisite if the software development process is ever to achieve the status of an engineering discipline." [Conte et al 86]

2.2 Software Reuse

A common method of problem solving is to apply a known solution to similar new problems. When the solution does not exactly fit the problem, the solution is adapted or extended. Proven solutions become accepted and standardized. These techniques apply to the world of software engineering as they do to everyday life. In software engineering, the reuse of software components is known to result in substantial quality and productivity payoffs [Agresti and Evanco 92][Card et al 86][Chen and Lee 93][Frakes 91]. Recent estimates of the quality and productivity payoffs from reuse fall between 10 and 90 percent [Frakes 91]. With reuse, software development becomes a capital investment.
With growing recognition that software reuse is economically viable, the market is demanding tools to assist the process of reuse. Most of these tools focus on the reuse of source code; methods of storing, searching, and retrieving source code components are becoming more common in development environments. Frakes defines software reuse as "the use of existing engineering knowledge or artifacts to build new systems" [Frakes 93]. Software reuse can apply to any product of the development life cycle, not only to fragments of source code. At each phase of the development process, developers should consider how previously completed work can be used to reduce the effort needed for the current task. This means that developers can pursue reuse of requirements documents, system specifications, design structures, and any other development artifact [Barnes and Bollinger 91]. Jones [Jones 93] identifies ten potentially reusable aspects of software projects:

1. architectures
2. source code
3. data
4. designs
5. documentation
6. estimates (templates)
7. human interfaces
8. plans
9. requirements
10. test cases

2.2.1 Software Reuse Terminology

Table 1 summarizes some types of software reuse that are defined in the research literature. While other references may precede the one mentioned in the description, information about each concept can be found in the listed reference. This list affirms the attention that reuse is currently receiving, and reveals the range of terms and definitions used to describe software reuse.
<table>
<thead>
<tr>
<th>Type of Reuse</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>Fenton [Fen91] defines public reuse as &quot;the proportion of a product which was constructed externally.&quot; See external.</td>
</tr>
<tr>
<td>private</td>
<td>Fenton [Fen91] defines private reuse as &quot;the extent to which modules within a product are reused within the same product.&quot; See internal.</td>
</tr>
<tr>
<td>external</td>
<td>External reuse level [Frakes 90] is the number of lower level items from an external repository in a higher level item divided by the total number of lower level items in the higher level item. See public.</td>
</tr>
<tr>
<td>internal</td>
<td>Internal reuse level [Frakes 90] is the number of lower level items not from an external repository which are used more than once divided by the total number of lower level items not from an external repository. See private.</td>
</tr>
<tr>
<td>verbatim</td>
<td>Bieman and Karunanithi define verbatim reuse as reuse of some item without modifications [Bieman and Karunanithi 93]. See black-box.</td>
</tr>
<tr>
<td>generic</td>
<td>Generic reuse is reuse of generic packages, such as templates for packages or subprograms [Bieman and Karunanithi 93].</td>
</tr>
<tr>
<td>leveraged</td>
<td>Bieman and Karunanithi define leveraged reuse as reuse with modifications [Bieman and Karunanithi 93].</td>
</tr>
<tr>
<td>black-box</td>
<td>Black-box reuse is the reuse of software components without any modification [Prieto-Diaz 93]. See verbatim.</td>
</tr>
<tr>
<td>white-box</td>
<td>White-box reuse is the reuse of components by modification and adaptation [Prieto-Diaz 93]. See leveraged.</td>
</tr>
<tr>
<td>direct</td>
<td>Direct reuse is reuse without going through an intermediate entity [BK].</td>
</tr>
<tr>
<td>indirect</td>
<td>Indirect reuse is reuse through an intermediate entity. The level of indirection is the number of intermediate entities between the reusing item and the item being reused [Bieman and Karunanithi 93].</td>
</tr>
<tr>
<td>adaptive</td>
<td>Adaptive reuse is a reuse strategy which uses large software structures as invariants and restricts variability to low-level, isolated locations. An example is changing arguments to parameterized modules [Barnes and Bollinger 91].</td>
</tr>
<tr>
<td>compositional</td>
<td>Compositional reuse is a reuse strategy which uses small parts as invariants; variant functionality links those parts together. Programming in a high level language is an example [Barnes and Bollinger 91].</td>
</tr>
<tr>
<td>vertical scope</td>
<td>Vertical reuse is reuse within the same application or domain. An example is domain analysis or domain modeling [Prieto-Diaz 93].</td>
</tr>
<tr>
<td>horizontal scope</td>
<td>Horizontal reuse is reuse of generic parts in different applications. Booch Ada Parts and other subroutine libraries are examples [Prieto-Diaz 93].</td>
</tr>
<tr>
<td>planned mode</td>
<td>Planned reuse is the systematic and formal practice of reuse as found in software factories [Prieto-Diaz 93].</td>
</tr>
<tr>
<td>ad-hoc mode</td>
<td>Ad-hoc reuse refers to the selection of components which are not designed for reuse from general libraries; reuse is conducted by the individual in an informal manner [Prieto-Diaz 93].</td>
</tr>
<tr>
<td>compositional</td>
<td>Compositional reuse is the use of existing components as building blocks for new systems. The Unix shell is an example [Prieto-Diaz 93].</td>
</tr>
<tr>
<td>generative</td>
<td>Generative reuse is reuse at the specification level with application or code generators. Generative reuse offers the &quot;highest potential payoff.&quot; The Refine and MetaTool systems are state of the art examples [Prieto-Diaz 93].</td>
</tr>
<tr>
<td>reuse-in-the-small</td>
<td>Reuse-in-the-small is the reuse of components which are dependent upon the environment of the application for full functionality. Favaro asserts that component-oriented reuse is reuse-in-the-small [Favaro 91].</td>
</tr>
<tr>
<td>reuse-in-the-large</td>
<td>Reuse-in-the-large is the use of large, self-contained packages such as spreadsheets and operating systems [Favaro 91].</td>
</tr>
</tbody>
</table>
The terms in the table describe various reuse issues. They address the quantity of reuse that occurs in a given product and methods of reuse implementation. Terms such as *reuse-in-the-large* and *reuse-in-the-small* provide categorization of the reused component. Some terms in the table overlap in meaning. For example, the terms *public* and *external* both describe the part of a product which was constructed externally; *private* and *internal* describe the part of a product which was not constructed externally but is developed and reused within a single product. The terms *verbatim* and *black-box* both describe reuse without modification; *leveraged* and *white-box* describe reuse with modification.

### 2.2.2 The Emerging Technology of Software Reuse

A software development environment supports the software development process. One part of a development environment might be a tool to assist the reuse of existing software components. Reuse libraries and classification systems are common functions within such a tool. Each reusable component must be efficiently stored, retrieved, and represented so that it can be found, understood, and integrated. In their paper "Representing reusable software," [Frakes and Gandel 90] propose a framework for software reuse representation and discuss methods of representing reusable components. The framework for reuse representation is intended to encompass any life-cycle object.

Frakes and Gandel group methods for representing reusable software into three categories: indexing languages from library and information science, knowledge-based methods from AI, and hypertext. Catalogs and indexes are traditionally used to provide searching mechanisms for software components that satisfy a need. Indexing languages
provide an interface to the search engine; it defines an item's location and summarizes its content. Frakes and Gandel extensively discuss the range of indexing languages now available and apply the languages to software reuse. Several knowledge-based methods of representations have developed from the field of Artificial Intelligence. Semantic nets, rules, and frames are discussed. Hypertext provides a structure to navigate text through a series of links. Several commercial software library retrieval systems incorporate hypertext technology.

Reusable component representation, storage and retrieval systems are being developed commercially and for research. At the University of Texas, for example, Brown, et. al. [Browne et al 90] have developed the Reusability-Oriented Parallel programming Environment (ROPE), a software component reuse system which helps a designer find and understand components using a classification method called structured relational classification. ROPE is integrated with a development environment called CODE (Computation-Oriented Display Environment), which supports construction of parallel programs using a declarative and hierarchical graph model of computation. ROPE supports reuse of both design and code components, focusing on the key issues of reusability: finding components, then understanding, modifying, and combining them.

Biggerstaff [Frakes 91] identifies the technologies that enable reuse: reuse libraries, classification systems, CASE tools, and object-oriented programming languages. The topic of reuse in an object-oriented environment is addressed in the next section of this paper. CASE tools provide a standardized environment to promote reuse. It should be pointed out that high technology within a reuse program is important but not essential to the success of the program [Frakes 91].
2.2.3 Software Reuse in the Object-Oriented Environment

In the above discussion, object-oriented programming languages is listed as one technology that enables reuse. Some researchers believe that the architecture of the object-oriented methodology increases reuse potential [McGregor and Sykes 92][Smith 90][Frakes 91]. They assert that aspects of the architecture such as classification, abstraction, and inheritance support and enable software reuse. Entire books are dedicated to the subject of effectively reusing software in the object-oriented environment [McGregor and Sykes 92][Smith 90]. A recent article by Prieto-Diaz states that "object orientation is seen as the technique of the future for reuse" [Prieto-Diaz 93]. Yet with all the speculation, empirical evidence of the benefits of object orientation for reuse is limited.

[McGregor and Sykes 92] identifies the following levels of reuse which can occur in the object-oriented paradigm:

- **Abstract-level reuse.** This is the use of high-level abstractions within an object-oriented inheritance structure as the foundation for new ideas or additional classification schemes.
- **Instance-level reuse.** Instance-level reuse is the most common form of reuse in an object-oriented environment. It is defined as simply creating an instance of an existing class.
- **Customization reuse.** This is the use of inheritance to support incremental development. A new application may inherit information from an existing class, overriding certain methods and adding new behaviors.
• *Source code reuse.* This is the low-level modification of an existing class to change its performance characteristics.

Several researchers identify aspects of the object-oriented paradigm which support software reuse [McGregor and Sykes 92][Smith 90][Owens 93]. Before addressing those aspects, it will be helpful to know the characteristics of a good design for reuse. The working group on Design for Reuse at the Fifth Annual Workshop on Institutionalizing Software Reuse lists the following characteristics of a good design for reuse [Griss and Tracz 93]:

1. *Strive to reuse black-box components, not cut-and-pasted source code.*
Black-box reuse is defined in Table 1 as "the reuse of software components without any modification." For effective reuse, design your system to reuse complete, unmodified software components.

2. *Identify, encapsulate, and specify commonalities and variabilities.*
The common attributes of reusable components should be identified and explicitly listed. Likewise, the differences also need to be recognized.

3. *Separate specification of abstract interface from implementation.*
Implementation details of a reusable component should be independent of the interface which a client will use to access the component. Thus implementation details can be modified or enhanced without altering the abstract interface. Abstract data types are an important part of this procedure.

4. *Do not allow a client to break a component's abstraction.*
Abstraction is a conceptual generalization which captures relevant aspects of a set of objects while leaving out irrelevant details [Krueger 92]. A client of a reusable component should be able to integrate the component with parameterization and extension without violating the abstract design.

5. *Extend component behavior by addition only, not by modification.*

The reusability of a component is enhanced if it can be adapted by adding functionality to the component without altering the original component's definition or implementation.

The literature identifies the following aspects of the object-oriented paradigm which support the above characteristics of a good design for reuse:

- **Encapsulation** - The class structure of object-oriented systems encapsulate data and procedures into functional components [Owens 93][Smith 90].

- **Abstraction** - Object-oriented languages provide for the development of a specification of a class that is separate from the implementation. This hides implementation details and supports reuse without requiring understanding of a specific class implementation [McGregor and Sykes 92].

- **Integration** - Owens [Owens 93] identifies two features of object-oriented methods that support integration of components. The inheritance mechanism specifies commonality and dependence between modules. Secondly, a *framework* is a generic class architecture which specifies the relationship between classes within a library or a family of applications [McGregor and Sykes 92].

- **Incremental development** - The object-oriented environment supports the enhancement of component behavior by addition rather than modification. A class does not
need to be fully implemented to be useful; evolutionary development is a natural tendency in the object-oriented paradigm [McGregor and Sykes 92].

Limited empirical evidence showing the benefits of the object-oriented paradigm toward software reuse does exist. In a case study by Owens [Owens 93], a C library of device I/O functions was used as a starting point in the comparison of procedural (C) code and object-oriented (C++) code. Owens states that the C library, containing over two hundred functions, is reasonably modular, and each module has fairly high cohesion.

The library was redesigned in the object-oriented paradigm and a C++ class library was implemented. Functions which were common to several modules were abstracted into parent classes, leading to increased internal reuse. To support the generic architecture, variabilities were encapsulated in parameters and virtual functions. Derived classes provided enhanced versions of base classes. The class library provided enhanced reuse via inheritance and "has proven easier to maintain than the C function library." Specific examples of C++ classes show improvements in flow control decisions, variable security, and encapsulation.

To measure the amount of reuse in the C library and the new C++ library, the reuse level measurement tool rl [Frakes 90] was run on both sets of code. The C++ code was measured by using cfront (a C++ to C translator from AT&T) to produce C code, and rl was run on the result. The results are summarized:
Table 2: Software Reuse in C vs. C++ [Owens 93]

<table>
<thead>
<tr>
<th></th>
<th>C library</th>
<th>C++ library</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Modules</td>
<td>10</td>
<td>12</td>
</tr>
<tr>
<td>Internal Functions Reused</td>
<td>68</td>
<td>80</td>
</tr>
<tr>
<td>External Functions Used</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>Total Functions Used</td>
<td>229</td>
<td>209</td>
</tr>
<tr>
<td>Internal Reuse Level</td>
<td>29%</td>
<td>38%</td>
</tr>
<tr>
<td>External Reuse Level</td>
<td>11%</td>
<td>12%</td>
</tr>
</tbody>
</table>

The results show an increase in the number of internal functions reused and a decrease in the total number of functions, leading to a higher internal reuse level. (See the discussion of [Frakes 90] in Section 2.3.1.3 for precise definitions of internal reuse level and the other terms in the table.) The C++ library has more functionality with fewer C functions, indicating that "internal reuse played a significant part in improving this library."

Another study of reuse in the object-oriented environment was conducted by Chen and Lee [Chen and Lee 93]. They developed an environment, based on an object-oriented approach, to design, manufacture and use reusable C++ components. A controlled experiment was conducted to substantiate the reuse approach in terms of software productivity and quality. Results showed improvements in software productivity. Measured in lines of code per hour (LOC/hr), productivity increased from 30 to 90% using the proposed construction approach. When the effort required to produce new code is greater than the effort of reusing reusable code, the benefits of reusability are in proportion to productivity.

The experiment conducted by Owens validates the improvement in software reuse in an object-oriented environment, showing more reuse in a C++ rewrite than in the original C
system. Chen and Lee validate the benefits of reuse in C++ in terms of quality and productivity. However, there is no substantial evidence that software reuse, along with additional quality and productivity, is significantly easier to obtain in the object-oriented paradigm. While theoretical speculations abound, empirical evidence is lacking.

2.2.4 Implementation of Software Reuse

According to Frakes [Frakes 92], "software reuse is widely believed to be the most promising technology for significantly improving software quality and productivity."

To be successful, however, research shows that implementation of a reuse program must be planned, deliberate, and systematic [Favaro 91] [Frakes 91] [Prieto-Diaz 93]. This section discusses some issues that an organization must address as it implements a reuse program. Reuse measurement is such an issue.

Table 3 shows the primary motivations for software reuse, the factors that affect it, and the reasons it may not succeed. This information is from [Frakes 93].
Table 3: Questions and Answers about Software Reuse Implementation [Frakes 93]

<table>
<thead>
<tr>
<th>Questions about software reuse implementation:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1. Why Try It?</strong></td>
<td><strong>Better early estimates</strong>&lt;br&gt;• Improved productivity&lt;br&gt;• Improved quality and reliability&lt;br&gt;• Better bid estimation</td>
</tr>
<tr>
<td><strong>2. What Issues Affect It?</strong></td>
<td><strong>Legal</strong>&lt;br&gt;• Managerial&lt;br&gt;• Economic</td>
</tr>
<tr>
<td><strong>3. Why Will It Not Succeed?</strong></td>
<td><strong>Part doesn't exist</strong>&lt;br&gt;• No attempt to reuse&lt;br&gt;• Part isn't available&lt;br&gt;• Part isn't understood&lt;br&gt;• Part can't be integrated</td>
</tr>
</tbody>
</table>

Section 2.1.4 discussed methods of modeling for the software development process. Frakes [Frakes 93] identifies the model for the reuse industrial experiment:

\[
\text{benefits} = f(\text{reuse level}) = g(\text{reuse factors}).
\]

The reuse benefits, improved quality and productivity, are a function of the reuse level, and the reuse level is a function of reuse factors. Reuse factors fall into four categories, identified in Table 3: managerial, economic, legal, and technical. Several sources show that management support is an essential ingredient of a successful reuse program [Frakes 93][Favaro 91][Frakes 91][Nunamaker and Chen 89]. Management must enforce policies that encourage standardization and component reuse as well as provide continuous education regarding components [Frakes 91]. The economic viability for software reuse must be created and maintained. The topic of measuring the success of reuse according to the degree of economic benefits will be addressed in Section 2.3.1.1. Legal issues regarding component creation and reuse by other organizations must also be addressed. And finally, technical issues such as reuse support tools and the reuse approach, parts-
based or formal-language-based, must be determined. A parts-based approach to reuse involves a programmer who integrates the software parts by hand. Domain-knowledge is encoded into an application generator in the formal-language-based approach [Frakes 93].

The reuse failure mode model was developed by Frakes to determine why reuse is not taking place in an organization [Frakes 90]. Failure mode analysis is the detailed examination of some failed product or process to determine why it failed, and to determine corrective action. The third question in Table 3 summarizes a set of failure modes for software reuse. The given modes are a first-level analysis of an interdependent set of causes for failure. At least two failure modes relate to the lack of education or training of software developers: part isn't understood and part can't be integrated. In two different case studies, insufficient training was found to be a barrier to reuse. Favaro [Favaro 91] reports that the limited knowledge of essential concepts such as abstraction and object-oriented design significantly contributed to the difficulty of integrating reusable components into applications. Schaefer [Frakes 91] agrees. He asserts that technology enabling reuse such as modular design and data abstraction is not exploited because developers are not able to effectively apply it.

Nunamaker and Chen [Nunamaker and Chen 89] assert that software developers face technical and social obstacles to the successful implementation of software reuse. The technical issues evolve around a development environment that effectively supports reusability. The social obstacles are:

1. Software developers must be willing to share software development knowledge.
2. Developers must be willing to use existing solutions.
3. Appropriate resource allocation must occur for the identification and development of common functions, utilities, and tools.

4. Measures to keep track of the reuse rates of various software components should be established to assist in reviewing and improving the reuse program.

Thus, reuse measurement is established as an important element of an effective reuse program. Measures of the amount of reuse and methods to obtain the measures must be defined so that the effectiveness of the reuse program can be monitored and evaluated. The remainder of this paper will focus on the measurement of software reuse, beginning with a review of the literature.

2.3 Software Reuse Measurement

Combining the philosophies of software measurement and software reuse, software reuse measurement is the quantitative measurement of the amount of reuse of some software artifact within a defined scope. Most models of reuse measurement are objective product metrics, measuring the amount of reuse within a software product. As shown in the next section, however, a reuse maturity model categorizes the reuse process. (See Sections 2.1.2 and 2.1.3 for more information regarding objective/subjective and product/process metrics.) Some measurements of software reuse exist, but few are actively used in industry. Most measurements are based on comparisons between the length or size of reused code and the size of newly written code in a software product.

The primary motivations for measuring software reuse are: [Frakes 92]
• to monitor progress of the amount of reuse over time in the quest for achieving goals;
• to provide a basis for determining the effects of reuse on software productivity and quality;
• to provide insight in developing software that is reusable;
• to determine the effects of particular actions on the amount of reuse.

2.3.1 Existing Software Reuse Metrics

Table 4 presents a summary of models for reuse metrics. The economic models measure reuse in terms of the economic costs and profits resulting from reuse. Maturity models categorize reuse programs according to a scale of labeled reuse levels. The reuse ratio models measure reuse by comparing the amount of reused software to the amount of newly developed software.
<table>
<thead>
<tr>
<th>Economic Models</th>
<th>Source</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cost/Productivity Models</td>
<td>Gaffney, Durek [Gaffney and</td>
<td>Simple model: Let C=cost of software development. R=proportion of reused code in the product. b=cost relative to that of all new code of</td>
</tr>
<tr>
<td></td>
<td>Durek 89]</td>
<td>incorporating reused code into the product. Then C=(b-1)R + 1 and productivity P=1/C. Cost of development model: Let E=cost of developing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>a reusable component relative to the cost of producing a component that is not to be reused. Let n be the number of uses over which code</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cost will be amortized. Then C (cost) is C=(b + E/n-1)R+1. Quality of investment (Q) is the ratio of reuse benefits (B) to reuse</td>
</tr>
<tr>
<td>Quality of Investment</td>
<td>Barnes, Bollinger [Barnes and</td>
<td>benefits (R): Q = B/R. If Q&lt;1 then the reuse effort resulted in a net loss. If Q&gt;1 then the investment provided good returns.</td>
</tr>
<tr>
<td></td>
<td>Bollinger 91]</td>
<td></td>
</tr>
<tr>
<td>Maturity Models</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reuse Maturity Model</td>
<td>Kolton, Hudson [Kolton and</td>
<td>Levels an organization proceeds through working toward effective software reuse: 1. Initial/Chaotic 4. Planned 2. Monitored 5. Ingrained</td>
</tr>
<tr>
<td></td>
<td>Hudson 91]</td>
<td>3. Coordinated</td>
</tr>
<tr>
<td>Reuse Capability Model</td>
<td>Software Productivity</td>
<td>The Software Productivity Consortium identifies four stages in the risk-reduction growth implementation model for software reuse: 1.</td>
</tr>
<tr>
<td>Reuse Ratio Models</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reuse Level</td>
<td>Frakes [Frakes 90]</td>
<td>Assume a system consists of parts where a higher level item is composed of lower level items. Let L = total number of lower level items in</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the higher level item, E = number of lower level items from an external source in the higher level item, I = number of lower level items</td>
</tr>
<tr>
<td></td>
<td></td>
<td>in the higher level item not from an external source, M = number of items not from external source used more than once. Then, External</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reuse Level = E / L  Internal Reuse Level = M / L  Total Reuse Level = E/L + M/L</td>
</tr>
<tr>
<td>Reuse Fraction</td>
<td>Agresti, Evanco [Agresti and</td>
<td>The variable FNEMC is defined as the fraction of new or extensively modified software units. FNEMC is the number of new components plus the</td>
</tr>
<tr>
<td></td>
<td>Evanco 92]</td>
<td>number of extensively modified components divided by the total number of components. FNEMC is equal to one minus the &quot;reuse fraction.&quot;</td>
</tr>
</tbody>
</table>

23
The following sections discuss each of these models in turn.

2.3.1.1 Economic Models

Cost/Productivity Models

Gaffney and Durek propose three cost and productivity models for software reuse [Gaffney and Durek 89]. The simple model shows the cost of reusing software components. The cost-of-development model builds upon the simple model by representing the cost of developing reusable components. The general model represents the effect of creating reusable components within a given development project.

Simple model: Let C be the cost of software development for a given product relative to all new code (for which C=1). R is the proportion of reused code in the product (R<=1). b is the cost relative to that for all new code of incorporating the reused code into the new product (b<=1). Then the relative cost for software development is

\[(\text{relative cost of all new code})(\text{proportion of new code}) + \]

\[(\text{relative cost of reused software})(\text{proportion of reused software}).\]

Then

\[ C = (1)(1-R) + (b)(R) \]

\[ = (b-1)R + 1 \]

and the corresponding relative productivity is

\[ P = 1 / C = 1 / ((b-1)R+1). \]

Notice that b is expected to be <= 1. If not, it would not be cost efficient to reuse software components. The size of b varies with the level of abstraction of the reusable
component. If the reusable component is source code, then one must go through the requirements, design, and testing phases in a new development project; the authors estimate $b=0.85$. If the reusable component is requirements, design, and code, then only the testing phase must be done and $b=0.08$.

*Cost of development model:* Let $E$ represent the cost of developing a reusable component relative to the cost of producing a component that is not to be reused. $E$ is expected to be $>1$. Let $n$ be the number of uses over which code cost will be amortized. The new value for $C$ (cost) incorporates these measures:

$$C = (b + E/n - 1)R + 1$$

The *general* model is not discussed in detail in the paper. Further discussions propose models for the effect of reuse on software quality (number of errors) and on software development schedules. While no numerical data is given, the authors state that trade-offs can occur between the proportion of reuse and the costs of developing and using reusable components. Using reusable software parts results in higher overall development productivity. Also, costs of building reusable parts must be shared across many users to achieve higher payoffs from software reuse.

Margono and Rhoads applied the cost of development model to assess the economic benefits of a reuse effort on a large-scale Ada project (the United States Federal Aviation Administration's Advanced Automation System (FAA/AAS)) [Margono and Rhoads 93]. The authors applied the model to various types of software categorized by the source (local, commercial, or public) and mode of reuse (verbatim or modified). The equation for $C$ in the reuse economics model was modified to reflect the different acquisition, development, and integration costs. Results show that the development cost for reuse is
often twice the development cost of non-reuse. The additional cost due to reuse during the detailed design phase of development is estimated to be 60%.

**Quality of Investment**

In their paper *Making software reuse cost effective*, Barnes and Bollinger [Barnes and Bollinger 91] examine the cost and risk features of software reuse and suggest an analytical approach for making good reuse investments. Reuse activities are divided into *producer* activities and *consumer* activities. Producer activities are reuse investments, or costs incurred while making one or more work products easier to reuse by others. Consumer activities are reuse benefits, or measures in dollars of how much the earlier reuse investment helped or hurt the effectiveness of an activity. The *total reuse benefit* can then be found by estimating the reuse benefit for all subsequent activities that profit from the reuse investment, including future activities.

The *quality of investment* (Q) is the ratio of reuse benefits (B) to reuse investments (R):

\[ Q = \frac{B}{R} \]

If Q is less than one for a reuse effort, then that effort resulted in a net financial loss. If greater than one, then the investment provided good returns. Three major strategies are identified for increasing Q: 1) increase the level of reuse, 2) reduce the average cost of reuse, and 3) reduce the investment needed to achieve a given reuse benefit.

Favaro [Favaro 91] utilized the model developed by Barnes, et. al. to analyze the economics of reuse. The following variables and formulas are relevant.
Table 5: Barnes' and Bollinger's economic investment model

<table>
<thead>
<tr>
<th>Variable</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>% of code contributed by reusable components</td>
</tr>
<tr>
<td>b</td>
<td>integration cost of reusable component as opposed to development cost</td>
</tr>
<tr>
<td>RC</td>
<td>relative cost of overall development effort</td>
</tr>
<tr>
<td>RP</td>
<td>relative productivity</td>
</tr>
<tr>
<td>E</td>
<td>relative cost of making a component reusable</td>
</tr>
<tr>
<td>N₀</td>
<td>payoff threshold value (all component development costs are recovered)</td>
</tr>
</tbody>
</table>

Formulas:

\[
RC = (1-R) + Rb \\
RP = 1/RC \\
RC = (b+E/N-1)R+1 \\
N₀ = E/(1-b)
\]

Favaro's research team estimated quantities for R and b. They found it difficult to estimate R, unclear whether actual source code should be measured or relative size of the load modules. Should the code size of a generic module be counted only once, or every time the module is instantiated and code is duplicated in the application? b was even more difficult to estimate: is cost measured in the amount of real time necessary to install the component in the application, and should the cost of learning be included?

Favaro developed a classification of BOOCH [Booch 87] components according to their relative complexity. The classification used by Favaro is:

**monolithic** Components were found to have a similar complexity in development and use, and could therefore be considered equivalent for this purpose. (stacks, queues)
polyolithic Components exhibited similar complexity regarding integration. (lists, trees)

graph The most complex component in the repository, the graph is an example of a nontrivial, domain-dependent reusable component.

menu, mask End-products of the project. They were developed as generalized, reusable components so were included in the study.

The above categories are listed in order or increasing complexity. Monolithic and polyolithic are classifications of standard BOOCH components. The graph is a BOOCH component in itself. Menus and masks are complex applications developed by Favaro from the BOOCH components.

The following table shows values for E, the relative cost of making a reusable component, b, the integration cost of a reusable component, and N_0, the payoff threshold value.

<table>
<thead>
<tr>
<th></th>
<th>E</th>
<th>b</th>
<th>N_0 for simple implementations</th>
<th>N_0 for complex implementations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monolithic</td>
<td>1.0</td>
<td>0.10</td>
<td>1.33</td>
<td>2.56</td>
</tr>
<tr>
<td>Polythic</td>
<td>1.2</td>
<td>0.15</td>
<td>1.69</td>
<td>3.40</td>
</tr>
<tr>
<td>Graph</td>
<td>1.6</td>
<td>0.25</td>
<td>2.56</td>
<td>5.72</td>
</tr>
<tr>
<td>Menu</td>
<td>1.9</td>
<td>0.30</td>
<td>3.25</td>
<td>7.81</td>
</tr>
<tr>
<td>Mask</td>
<td>2.2</td>
<td>0.40</td>
<td>4.40</td>
<td>12.97</td>
</tr>
</tbody>
</table>

The overall costs of a reusable component relative to a non-reusable component is E+b. b is expected to be less than 1.0 since reusable components should be more easily
integrated. E is greater than or equal to 1.0, showing costs of developing reusable components are higher than costs of developing non-reusable components. The results show that the cost of reusability increased as the complexity of the component increased. Monolithic components were so simple there was essentially no extra cost to develop them as reusable components. In contrast, the cost of the mask component more than doubled as it was generalized. The integration cost b was also high in complex applications. The values for N0 show that the monolithic and polythetic components are amortized after only 2 uses. However, the graph component must be used approximately 5 times before its costs are recovered, and the most complex form of the mask will require 12.97 projects for amortization. In summary, the results show that as components of some size and complexity are developed for reuse, the costs rise quickly.

2.3.1.2 Maturity Models

Rather than providing a specific quantitative measurement of the amount of reuse in software, reuse maturity models identify the progression of reuse activities within an organization. A reuse maturity model categorizes a reuse program according to a scale of labeled reuse levels. The maturity model is at the core of planned reuse, helping organizations understand their past, current, and future goals for reuse activities [Prieto-Diaz 93].

Reuse Maturity Model

Kolton and Hudson [Kolton and Hudson 91] developed a maturity framework with five levels:

1. Initial/Chaotic
2. Monitored

3. Coordinated

4. Planned

5. Ingrained

They identified ten dimensions of reuse maturity; for each, an attribute was specified for each maturity level. The resulting matrix is shown in Table 7.
### Table 7: Hudson and Koltun Reuse Maturity Model

<table>
<thead>
<tr>
<th>Motivation/ Culture</th>
<th>1 Initial/ Chaotic</th>
<th>2 Monitored</th>
<th>3 Coordinated</th>
<th>4 Planned</th>
<th>5 Ingrained</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reuse discouraged</td>
<td>Reuse encouraged</td>
<td>Reuse incentivized re-enforced rewarded</td>
<td>Reuse incentivized rewarded</td>
<td>Reuse is the way we do business</td>
<td></td>
</tr>
<tr>
<td>Planning for reuse</td>
<td>None</td>
<td>Grassroots activity</td>
<td>Targets of opportunity</td>
<td>Business imperative</td>
<td>Part of strategic plan</td>
</tr>
<tr>
<td>Breadth of reuse</td>
<td>Individual</td>
<td>Work group</td>
<td>Department</td>
<td>Division</td>
<td>Enterprisewide</td>
</tr>
<tr>
<td>Responsible for making reuse happen</td>
<td>Individual initiative</td>
<td>Shared initiative</td>
<td>Dedicated individual</td>
<td>Dedicated group</td>
<td>Corporate group with division liaisons</td>
</tr>
<tr>
<td>Process by which reuse is leveraged</td>
<td>Reuse process chaotic; unclear how reuse comes in</td>
<td>Reuse questions raised at design reviews (after the fact)</td>
<td>Design emphasis placed on off the shelf parts</td>
<td>Focus on developing families of products</td>
<td>All software products are generalized for future reuse</td>
</tr>
<tr>
<td>Reuse assets</td>
<td>Salvage yard (no apparent structure to collection)</td>
<td>Catalog identifies language and platform specific parts</td>
<td>Catalog organized along application specific lines</td>
<td>Catalog includes generic data processing functions</td>
<td>Planned activity to acquire or develop missing pieces in catalog</td>
</tr>
<tr>
<td>Classification activity</td>
<td>Informal, individualized</td>
<td>Multiple independent schemes for classifying parts</td>
<td>Single scheme catalog published periodically</td>
<td>Some domain analyses done to determine categories</td>
<td>Formal, complete, consistent timely classification</td>
</tr>
<tr>
<td>Technology support</td>
<td>Personal tools, if any</td>
<td>Many tools, but not specialized for reuse</td>
<td>Classification aids and synthesis aids</td>
<td>Electronic library separate from development environment</td>
<td>Automated support integrated with development environment</td>
</tr>
<tr>
<td>Metrics</td>
<td>No metrics on reuse level, pay-off, or costs</td>
<td>Number of lines of code used in cost models</td>
<td>Manual tracking of reuse occurrences of catalog parts</td>
<td>Analyses done to identify expected payoffs from developing reusable parts</td>
<td>All system utilities, software tools and accounting mechanisms instrumented to track reuse</td>
</tr>
<tr>
<td>Legal, contractual, accounting considerations</td>
<td>Inhibitor to getting started</td>
<td>Internal accounting scheme for sharing costs and allocating benefits</td>
<td>Data rights and compensation issues resolved with customer</td>
<td>Royalty scheme for all suppliers and customers</td>
<td>Software treated as key capital asset</td>
</tr>
</tbody>
</table>

Notice that for each of the ten aspects of reuse, the amount of organizational involvement and commitment increases as the level progresses from initial/chaotic reuse to ingrained reuse. Ingrained reuse incorporates fully automated support tools and accurate reuse measurement to track progress.
Reuse Capability Model

The reuse capability model developed by the Software Productivity Consortium [Davis 93] identifies four stages in the implementation model for reuse:

1. *Opportunistic* The reuse strategy is developed on the project level. Specialized reuse tools are used and reusable assets are identified.

2. *Integrated* A standard reuse strategy is defined and integrated into the corporation's software development process. The reuse program is fully supported by management and staff. Reuse assets are categorized.

3. *Leveraged* The reuse strategy expands over the entire life cycle and is specialized for each product line. Reuse performance is measured and weaknesses of the program identified.

4. *Anticipating* New business ventures take advantage of the reuse capabilities and reusable assets. High payoff assets are identified. The reuse technology is driven by customer's needs.

2.3.1.3 Reuse Ratio Models

Reuse Level

In [Frakes 90], Frakes states that the basic dependent variable in software reuse measurement is the level of reuse. This is a parts-based approach to reuse measurement, assuming that a system is composed of parts which exist at different levels. Frakes states
that levels of abstraction must be defined to measure reuse. The following quantities can be calculated given a higher level item composed of lower level items:

\[ L = \text{the total number of lower level items in the higher level item.} \]

\[ E = \text{the number of lower level items from an external repository in the higher level item.} \]

\[ I = \text{the number of lower level items in the higher level item which are not from an external repository.} \]

\[ M = \text{the number of items not from an external repository which are used more than once.} \]

Given these quantities, the following reuse level metrics are proposed:

External Reuse Level = \( \frac{E}{L} \)

Internal Reuse Level = \( \frac{M}{L} \)

The user must provide some information to calculate the reuse measures. The user must define the abstraction hierarchy, a definition of external repositories, and a definition of the "uses" relationship. For each part in the parts-based approach, we must know the name of the part, source of the part (internal or external), the level of abstraction, and the amount of usage.

The tool \textit{rl} was built to perform reuse analysis of C code [Frakes 92]. With "system" as the higher level component and "function" as the lower, a C system can be broken down into functions which are internal or external within a calling hierarchy. \textit{rl} uses \textit{cflow}, a Unix tool, to produce calling hierarchy information. \textit{rl} was run on 29 systems and the resulting data includes internal reuse levels, external reuse levels, total reuse levels, and NCSL (non commentary source lines). The results show an average reuse level of 58%, a very high figure. DeMarco, for example, estimated 5% reuse on an average project [DeMarco and Lister 84]. The high level of reuse is partially attributed to the design of
the C programming language, with many simple system capabilities designed as functions. Internal reuse, however, is also high, at 7%.

A high correlation is shown between software size and reuse level. The correlation of external reuse vs. log NCSD is $r=-0.76$, meaning that high external reuse is directly related to small size. In contrast, internal reuse statistics also show a strong correlation with size but in the opposite direction, meaning that internal reuse is high in large software systems.

[Frakes and Arnold 90] presents a formal model of the reuse level metrics. Nodes and relationships are used to decompose a work product. A graphic hierarchical model contains nodes, represented by small circles, and two possible relationships between nodes which are represented by arrows. The possible relationships are $\text{depends\_on}$, denoting usage, and $\text{directly\_contains}$, denoting containment. A node represents a life cycle object. The authors assert that the model can be extended with additional object types, relationship types, and attributes.

An example maps objects from the Ada user domain to the formal model. The authors have created tables to aid the mapping process. The example shows how users can apply an actual domain to the formal model to obtain a reuse measurement model. A graphical catalog of metrics is available to users so that they can effectively use the model to create new metrics. The paper discusses the methods that users can use to implement reuse measurement through decomposition and mapping to the formal model.
Reuse Fraction

Proposed by Agresti and Evanco [Agresti and Evanco 92], the reuse fraction is a simple proportional metric which defines the fraction of reused compilation units. Compilation units are categorized into two classes:

1) those that are reused verbatim or with "slight" modification (\(\leq 25\%\) of lines changed),
2) those that are new or extensively modified (\(> 25\%\) of source lines changed).

Given the following variables,

\[
cun = \text{number of compilation units of newly developed code},
\]

\[
cux = \text{number of compilation units extensively modified},
\]

\[
cutot = \text{total number of compilation units},
\]

the variable FNEMC is defined as the fraction of compilation units that are new or extensively modified:

\[
\text{FNEMC} = \frac{\text{cun} + \text{cux}}{\text{cutot}}.
\]

FNEMC is equal to one minus the reuse fraction.

The reuse fraction measurement is specific to measuring reuse of source code compilation units. It is a simple, non-extensible variation of reuse level, discussed above.

2.3.1.4 Reuse Measurement in the Object-Oriented Environment

Bieman and Karunanithi [Bieman 92][Bieman and Karunanithi 93] have proposed reuse measurements which are specific to the object-oriented environment. [Bieman 92] identifies three perspectives from which to view reuse: the server perspective, the client perspective, and the system perspective. The server perspective is the perspective of the library or a particular library component, the analysis focusing on how the entity is reused
by the clients. From the client perspective, the goal is knowing how a particular program
entity reuses other program entities. The system perspective is a view of reuse in the
overall system, including servers and clients.

The server reuse profile of a class will characterize how the class is reused by the client
classes. The verbatim server reuse in an object oriented system is basically the same as in
procedural systems, using object oriented terminology. Leveraged server reuse is
supported through inheritance. A client can reuse the server either by extension, adding
methods to the server, or by overload, redefining methods. *(Note: [McGregor and Sykes
92] offers good definitions of the object-oriented terminology used in this section.)*

The client reuse profile characterizes how a new class reuses existing library classes. It
too can be verbatim or leveraged, with similar definitions to the server perspective.

Measurable system reuse attributes include:

- % of new system source text imported from the library
- % of new system classes imported verbatim from the library
- % of new system classes derived from library classes and the average % of the
  leveraged classes that are imported
- average number of verbatim and leveraged clients for servers, and servers for
  clients
- average number of verbatim and leveraged indirect clients for servers, and
  indirect servers for clients
- average length and number of paths between indirect servers and clients for
  verbatim and leveraged reuse
In [Bieman and Karunanithi 93], Bieman and Karunanithi describe a prototype tool which is under development to collect the proposed measures from Ada programs. This work recognizes the differences between object oriented systems and procedural systems and exploits those differences through unique measurements.

2.3.2 Measuring Software Reuse Potential

The above discussions address measuring the amount of software reuse. There is also work being done to measure the reusability of software: given a piece of software, how much effort must be exerted to reuse it?

Major work in this area is by Basili, et. al. [Basili et al 90] at the University of Maryland. Two reuse studies were performed with respect to the development and reuse of systems written in the Ada language. The first study defines a means of measuring data bindings to characterize and identify reusable components. The data bindings within a program are identified, and a cluster analysis is performed to identify which modules are strongly coupled and may not be good candidates for reuse, and which modules are found to be independent of others and are potentially reusable. Through application of these metric and analysis techniques, a set of guidelines are derived and listed for designing and building reusable Ada components.

The second study defines an abstract measurement of reusability of Ada software components. Potentially reusable software is identified, and a method to measure distances from that ideal is defined. By measuring the amount of transformation which must be performed to convert an existing program into one composed of maximally
reusable components, an indication of the reusability of the program can be obtained. The latent non-reusability of software can also be found by identifying transformations that cannot be performed cost effectively.

2.3.3 Relation of Reuse to Quality and Productivity

Since systematic software reuse is not common, empirical evidence relating software reuse to quality and productivity is limited. However, several researchers have accumulated and published statistics that support the notion that software reuse improves quality and productivity.

Agresti and Evanco [Agresti and Evanco 92] conducted a study to predict defect density (a software quality measurement) based on characteristics of Ada designs. Data used in the analysis, from the Software Engineering Laboratory (SEL) of NASA Goddard Space Flight Center, consists of 16 subsystems. The SEL project database provides data on the extent of reuse and subsystem identification for each compilation unit as well as reported defects and nondefect modifications. Collectively, approximately 149 KSLOC (kilo-source lines of code) were considered for the analysis. The project database showed that the reuse ratios (fraction of compilation units reused verbatim or with slight modification, \( \leq 25\% \) of lines changed) lie between 26 and 28\%. Defect density is between 3.0 and 5.5 total defects per KSLOC. Four sample rows from a table summarizing the project characteristics of the subsystems show that a high level of reuse correlates with a low defect density (size is in KSLOC units):
Table 8: Characteristics of SEL subsystems [Agresti and Evanco 92]

<table>
<thead>
<tr>
<th>Subsystem</th>
<th>Software size</th>
<th>Library units</th>
<th>Compilation units</th>
<th>Reuse</th>
<th>Defect Density</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-5</td>
<td>27.3</td>
<td>38</td>
<td>185</td>
<td>0.44</td>
<td>6.2</td>
</tr>
<tr>
<td>1-6</td>
<td>5.7</td>
<td>18</td>
<td>73</td>
<td>0.94</td>
<td>1.6</td>
</tr>
<tr>
<td>2-4</td>
<td>6.9</td>
<td>23</td>
<td>60</td>
<td>0.74</td>
<td>1.4</td>
</tr>
<tr>
<td>3-3</td>
<td>3.5</td>
<td>12</td>
<td>66</td>
<td>0.09</td>
<td>8.0</td>
</tr>
</tbody>
</table>

The Reusability-Oriented Parallel programming Environment (ROPE) [Browne et al 90] was briefly described in section 2.2.2. ROPE is integrated with a development environment called CODE (Computation-Oriented Display Environment), which supports construction of parallel programs using a declarative and hierarchical graph model of computation. ROPE supports reuse of both design and code components, focusing on the key issues of reusability: finding components, then understanding, modifying, and combining them. An experiment was conducted to investigate user productivity and software quality for the CODE programming environment, with and without the ROPE reusability system. The experimental design included metrics such as fraction of code in a program consisting of reused components, development time and error rates. Reuse rates were reported as "extremely high" for the 43 programs written using ROPE, with a mean reuse rate for a total program (code and design) equal to 79%. The researchers used total development time to measure the effect of reusability on productivity. Table 9 shows the development time in hours for subjects programming in the CODE environment and those programming in CODE and ROPE. The data reveals that ROPE had a significant effect on development time for all of the experimental programs.
Table 9: Mean Development Time and 95% Confidence Intervals in Hours [Browne et al 90]

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Using CODE only</th>
<th>Using CODE and ROPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convex Hull</td>
<td>12.4 [9.0,15.8]</td>
<td>2.2 [2.0,2.5]</td>
</tr>
<tr>
<td>Readers/ Writers</td>
<td>4.7 [3.4,6]</td>
<td>1.8 [1.2,2.4]</td>
</tr>
<tr>
<td>Producer/ Consumer</td>
<td>3.9 [3.6,4.3]</td>
<td>1.9 [1.2,8]</td>
</tr>
<tr>
<td>Shortest Path</td>
<td>33.3 [16,51]</td>
<td>1.4 [0.7,2.1]</td>
</tr>
<tr>
<td>Parallel Prefix</td>
<td>20 N/A</td>
<td>1.4 [0.9,1.8]</td>
</tr>
<tr>
<td>Divide Region</td>
<td>20 N/A</td>
<td>3.5 [2.5,4.5]</td>
</tr>
<tr>
<td>Sort/ Merge</td>
<td>8.5 N/A</td>
<td>1.5 N/A</td>
</tr>
</tbody>
</table>

Error rates were used to measure quality. Compile errors, execution errors, and logic errors were all counted. The results are shown in Table 10. The use of ROPE reduced error rates, but the data is less clear than that for productivity. The researchers attribute this to the difficulty of collecting the data and to the lack of distinction between design and code errors.

Table 10: Mean Number of Errors and 95% Confidence Intervals [Browne et al 90]

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Using CODE only</th>
<th>Using CODE and ROPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convex Hull</td>
<td>8.8 [4.3,13.3]</td>
<td>3.1 [1.4,4.4]</td>
</tr>
<tr>
<td>Readers/ Writers</td>
<td>14.5 [5.4,23.6]</td>
<td>1.2 [.4,2.0]</td>
</tr>
<tr>
<td>Producer/ Consumer</td>
<td>4.3 [1.9,6.7]</td>
<td>.3 [0,.7]</td>
</tr>
<tr>
<td>Shortest Path</td>
<td>10 N/A</td>
<td>4 N/A</td>
</tr>
<tr>
<td>Parallel Prefix</td>
<td>20 N/A</td>
<td>2.5 N/A</td>
</tr>
<tr>
<td>Divide Region</td>
<td>5 N/A</td>
<td>17 N/A</td>
</tr>
<tr>
<td>Sort/ Merge</td>
<td>7 N/A</td>
<td>3 N/A</td>
</tr>
</tbody>
</table>
In summary, the final results of the CODE/ROPE experimentation show a high correlation between the measures of reuse rate, development time, and decreases in number of errors.

In a relatively early study, Card, Church, and Agresti [Card et al 86] conducted an empirical study of software design practices in a Fortran-based scientific computing environment. The goals of the analysis of software reuse were to identify the types of software that are reused and to quantify the benefits of software reuse. The results were:

- The modules that were reused without modification tended to be small and simple, exhibiting a relatively low decision rate.
- Extensively modified modules tended to be the largest of all reused software (rated from extensively modified to unchanged) in terms of the number of executable statements.
- 98 percent of the modules reused without modification were fault free and 82 percent of them were in the lowest cost per executable statement category.
- These results were consistent with a previous Software Engineering Laboratory study [Card et al 82] which shows that reusing a line of code costs only 20 percent of the cost of developing it new.

Kazuo Matsumura was a panelist at the International Conference on Software Engineering forum entitled Software Reuse: Is It Delivering? [Frakes 91]. In the paper, Matsumura describes an implementation of a reuse program. Results of the reuse program implementation show a 60% ratio of reuse components and a decrease in errors by 20 to 30%. Managers felt that the reuse program would be profitable if a component were reused at least three times.
The Cost/Productivity Model by Gaffney and Durek [Gaffney and Durek 89] was discussed in Section 2.3.1.1. The models specify the effect of reuse on software quality (number of errors) and on software development schedules. Results in the paper suggest that trade-offs can occur between the proportion of reuse and the costs of developing and using reusable components. In a study of the latent error content of a software product, the relative error content decreased for each additional use of the software but leveled off between 3 and 4 uses. The models show that the number of uses of the reusable software components directly correlates to the development product productivity. The authors believe that the costs of building reusable parts must be shared across many users to achieve higher payoffs from software reuse.

2.4 Literature Review Summary

This literature search discusses many aspects of software reuse and software reuse measurement. The topic is an active field in the research community, perhaps because, as shown above, software reuse is an effective method of increasing software quality and productivity and thus reducing the costs of software development. However, as also pointed out in the review, developing and using reusable software does have its risks and costs. The organization must be willing to plan and support the reuse effort. [Margono and Rhoads 93] states that the development cost for reusable software is often twice the cost of developing non-reusable software. In the case study by [Favaro 91], the development costs of reusable components were amortized after 2 to 13 uses, depending on the complexity of the component. On the benefits side of the reuse costs equation, [Card et al 86] shows that reusing a line of code is only one-fifth the cost of developing it.
Software reuse has been shown to increase quality and productivity. In [Agresti and Evanco 92], high levels of reuse were shown to result in low defect densities. [Card et al 86] found that 98% of the modules reused without modification were fault-free. Software reuse resulted in a decrease in errors by 20 to 30% in [Frakes 91]. In [Gaffney and Durek 89], the latent error content for a reused module leveled off at 0.30 (relative to all new code) after three to four uses. [Browne et al 90] showed conclusive evidence that software development time was significantly reduced in an environment supporting reuse, but empirical studies supporting productivity gains are few. The productivity gains for reuse seem to be taken for granted; using a pre-existing component that can be easily integrated into an application naturally requires less time than developing the component from scratch.

The literature review has delineated the need and benefits of measuring software reuse. While the cost/productivity models formally measure the costs and benefits of reuse, Frakes is the only researcher who proposes a formal model for reuse ratio measurement. The remainder of this paper expands upon the initial research performed by Frakes as summarized in Section 2.3.1.3.
3 The Reuse Metrics

3.1 Introduction

Frakes introduced the reuse level metric in [Frakes 90], stating that the basic dependent variable in software reuse measurement is the level of reuse. This parts-based approach to reuse measurement assumes that a system is composed of parts at different levels of abstraction. For example, a C system is composed of functions, and functions are composed of lines of code. The levels of abstraction must be defined to measure reuse. The reuse level of a C system could be defined in terms of functions; in this case, the higher level component is a system and the lower level component is a function. The reuse level of a function could be expressed in terms of lines of code, in which case the higher level component is a function and the lower level component is a line of code.

A software component (lower level item) may be internal or external. An internal lower level component was developed for the higher level component. An external lower level component is used by the higher level component but was created for a different item or for general use.

The definition of reuse may vary. A traditional definition for a reused function within a C system is one that is called from more than one place within the system. Alternatively, the user of the metric may wish to define reuse after an arbitrary number of calls. Also, rather than counting the number of places that call a reused function, one may want to count the number of actual calls.
3.2 Definition of Reuse Level

Given a higher level item composed of lower level items, reuse level metrics may be defined. [Frakes and Arnold 90] defines the internal reuse level of a higher level item as the number of reused internal lower level items divided by the total number of lower level items in the higher level item. The external reuse level of a higher level item is the number of reused external lower level items in the higher level item divided by the total number of lower level items in the higher level item. The total reuse level is the sum of internal reuse level and external reuse level.

I have extended the reuse level metric to take into consideration a reuse threshold level. The internal threshold level is the maximum number of uses of an internal item that can occur before reuse occurs. The external threshold level is the maximum number of uses of an external item that can occur before reuse occurs. The variables and reuse level metrics are:

\[
\begin{align*}
\text{ITL} &= \text{internal threshold level, the maximum number of uses of an internal item that can occur before reuse occurs.} \\
\text{ETL} &= \text{external threshold level, the maximum number of uses of an external item that can occur before reuse occurs.} \\
\text{IU} &= \text{number of internal lower level items which are used more than ITL.} \\
\text{EU} &= \text{number of external lower level items which are used more than ETL.}
\end{align*}
\]
\[ T = \text{total number of lower level items in the higher level item, both internal and external.} \]

Internal reuse level: \[ IU / T \]

External reuse level: \[ EU / T \]

Total reuse level: \[ (IU + EU) / T \]

Internal, external, and total reuse level will assume values between 0 and 1:

\[ 0 \leq \text{Internal reuse level} \leq 1 \]

\[ 0 \leq \text{External reuse level} \leq 1 \]

\[ 0 \leq \text{Total reuse level} \leq 1. \]

More reuse occurs as the reuse level value approaches 1. A reuse level of 0 indicates no reuse.

One requirement for the calculation of the reuse level metrics is a definition of the uses relationship which holds between a component and any other component it references. A call graph abstraction may be employed to illustrate how lower level items are used within a higher level item. Figure 1 shows a call graph abstraction for higher level item H. Each node represents a lower level item. The label 'I' indicates an internal item and 'E' represents an external item. The directional arc between nodes represents the uses relationship, or a reference.
A simple algorithm can be defined to calculate the value of each variable in the reuse level equations. Given a call graph abstraction such as the one shown above, the algorithms for IU, EU and T are:

To calculate IU:

1. set inode_cnt = 0
2. for each node labeled 'I', called x:
   if number of references to x > ITL then
     inode_cnt = inode_cnt + 1
3. IU = inode_cnt
To calculate EU:

1. set enode_cnt = 0
2. for each node labeled 'n', called y:
   
   if number of references to y > ETL then
   
   enode_cnt = enode_cnt + 1

3. EU = enode_cnt

To calculate T:

1. set node_cnt = 0
2. for each node:
   
   node_cnt = node_cnt + 1

3. T = node_cnt

Given values for the variables IU, EU and T, the values for internal, external and total reuse level can be easily computed using the formulas on page 46. Table 11 shows the values for the reuse level metric for the higher level item H shown in Figure 1.

<table>
<thead>
<tr>
<th></th>
<th>ITL=1</th>
<th>ITL=2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ETL=0</td>
<td>ETL=1</td>
</tr>
<tr>
<td>IU</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>EU</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>T</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Internal reuse level</td>
<td>1/7</td>
<td>0</td>
</tr>
<tr>
<td>External reuse level</td>
<td>3/7</td>
<td>1/7</td>
</tr>
<tr>
<td>Total reuse level</td>
<td>4/7</td>
<td>1/7</td>
</tr>
</tbody>
</table>

The first column in Table 11 contains the values for reuse level using an internal threshold level of one and an external threshold level of zero. This means that at least two references to an internal node constitutes reuse. Only one reference to an external node constitutes reuse. The second column uses an internal threshold level of two and an
external threshold level of one. As expected, the reuse level values are less in the second column.

A different definition of the uses relationship may allow a node to reference another node more than once. For example, a single call might be regarded as a single reference, and additional calls are distinct and counted. Figure 2 is a multSCALL graph abstraction. Each arc is labeled with a digit indicating the number of references.

![MultSCALL Graph Abstraction for Higher Level Item H](image)

*Figure 2: MultSCALL Graph Abstraction for Higher Level Item H*

The algorithms shown above for IU, EU and T can be used to calculate corresponding values for a multSCALL graph abstraction. In the multSCALL graph abstraction, however, the
digit which labels each arc must be taken into consideration. If, for example, an arc is labeled with a 3, then that single arc represents three references to the node. Table 12 shows the values for the reuse level metric for the higher level item H shown in Figure 2.

<table>
<thead>
<tr>
<th></th>
<th>ITL=1 ETL=0</th>
<th>ITL=2 ETL=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>IU</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>EU</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>T</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Internal reuse level</td>
<td>2/7</td>
<td>1/7</td>
</tr>
<tr>
<td>External reuse level</td>
<td>3/?</td>
<td>1/7</td>
</tr>
<tr>
<td>Total reuse level</td>
<td>5/7</td>
<td>2/7</td>
</tr>
</tbody>
</table>

Table 12: Reuse level values for the multcall graph abstraction shown in Figure 2.

3.3 Related Metrics

The internal reuse level, external reuse level, and total reuse level metrics were initially defined by Frakes in [Frakes 90]. This project and report proposes further metrics which build upon and enhance the original reuse level metrics.

3.3.1 Reuse Frequency

Referring to Figure 1, each directional arc between nodes represents the uses relationship, or a reference. The reuse frequency metric is based on references to reused components rather than on the components themselves. The internal reuse frequency of a higher level item is the number of references to reused internal lower level items divided
by the total number of references in the higher level item. The external reuse frequency of a higher level item is the number of references to reused external lower level items divided by the total number of references in the higher level item.

The variables and reuse frequency metrics are:

\[ IUF = \text{number of references in the higher level item to reused internal lower level items.} \]

\[ EUF = \text{number of references in the higher level item to reused external lower level items.} \]

\[ TF = \text{total number of references to lower level items in the higher level item, both internal and external.} \]

*Internal reuse frequency:* \( IUF / TF \)

*External reuse frequency:* \( EUF / TF \)

*Total reuse frequency:* \( (IUF + EUF) / TF \)

Internal, external, and total reuse frequency will assume values between 0 and 1:

\[ 0 \leq \text{Internal reuse frequency} \leq 1 \]

\[ 0 \leq \text{External reuse frequency} \leq 1 \]

\[ 0 \leq \text{Total reuse frequency} \leq 1. \]

Again, algorithms can be defined to calculate the value of each variable in the reuse frequency equations. The algorithms for IUF, EUF and TF are shown below:

To calculate IUF:

1. set \( \text{iref}_\text{cnt} = 0 \)
2. for each node labeled 'I', called \( x \):
if number of references to x > ITL then
  for each reference to x:
    iref_cnt = iref_cnt + 1
3. IUF = iref_cnt

To calculate EUF:
1. set eref_cnt = 0
2. for each node labeled 'z', called y:
   if number of references to y > ETL then
      for each reference to y:
        eref_cnt = eref_cnt + 1
3. EUF = eref_cnt

To calculate TF:
1. set ref_cnt = 0
2. for each arc:
   ref_cnt = ref_cnt + 1
3. TF = ref_cnt

Given values for the variables IUF, EUF and TF, the values for internal, external and total reuse frequency can be computed using the formulas on page 51. Table 13 shows the values for the reuse frequency metric for the higher level item H shown in Figure 1.
Table 13: Reuse frequency values for Figure 1.

<table>
<thead>
<tr>
<th></th>
<th>ITL=1</th>
<th>ETL=2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ITL=1</td>
<td>ETL=2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IUF</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>EUF</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>TF</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Internal reuse frequency</td>
<td>2/9</td>
<td>0</td>
</tr>
<tr>
<td>External reuse frequency</td>
<td>5/9</td>
<td>3/9</td>
</tr>
<tr>
<td>Total reuse frequency</td>
<td>7/9</td>
<td>3/9</td>
</tr>
</tbody>
</table>

Table 14 shows the reuse frequency values for H using a multical graph abstraction as shown in Figure 2.

Table 14: Reuse frequency values for the multical graph abstraction shown in Figure 2.

<table>
<thead>
<tr>
<th></th>
<th>ITL=1</th>
<th>ETL=2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ITL=1</td>
<td>ETL=2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IUF</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>EUF</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>TF</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>Internal reuse frequency</td>
<td>5/12</td>
<td>3/12</td>
</tr>
<tr>
<td>External reuse frequency</td>
<td>6/12</td>
<td>4/12</td>
</tr>
<tr>
<td>Total reuse frequency</td>
<td>11/12</td>
<td>7/12</td>
</tr>
</tbody>
</table>

3.3.2 Complexity Weighting

A weighting has been implemented to indicate the complexity of a reused component. Program size is often used as a measure of complexity [Conte et al 86]. The complexity
weighting for reuse can assume multiple definitions. This research defines the complexity weighting for internal reuse as the sum of the sizes of all reused internal lower level items divided by the sum of the sizes of all internal lower level items within the higher level item. A different approach might define the complexity weighting as the average size of all reused components relative to the average size of all components that are not reused.

To calculate the complexity weighting, the following information is needed:

- A definition of higher and lower level items,
- A measure of size for the lower level items,
- Which lower level items are reused within the higher level item.

An example complexity weighting for internal reuse in a C system is the ratio of the size (calculated in number of lines of non-commentary source code) of reused internal functions to the size of all internal functions in the system.
4 The rl Software

4.1 Overview

Frakes built the tool rl to perform reuse analysis of C code [Frakes 92]. In its original form, rl reported the following metrics for C code:

1. internal reuse level - the number of internal functions used more than once in a given set of C files (a C system) divided by the total number of functions in the system.
2. external reuse level - the number of external functions used within a given set of C files divided by the total number of functions.
3. total reuse level - the sum of internal reuse level and external reuse level.

I have extended rl to perform a more rigorous reuse analysis of C code along with more flexibility and better reporting. This Chapter describes in detail the rl software. A listing of the software can be found in Appendix B. The manual page in Appendix C provides instructions for using rl.

4.1.1 Purpose of the rl Software

As stated above, the rl software performs reuse analysis of C source code. Given a set of C files, rl reports the following information:
1. internal reuse level
2. external reuse level
3. total reuse level
4. internal reuse frequency
5. external reuse frequency
6. total reuse frequency
7. complexity (size) weighting for internal functions

The rl program accepts parameters which specify the internal threshold level and external threshold level. The default values for these arguments are 1 and 0, respectively. The user may request usage of a multical graph abstraction, in which each call to a function is considered a "use." The rl program also allows multiple definitions of higher level and lower level abstractions. The allowed higher level abstractions are system, file or function. The lower level abstraction may be function or NCSL (Non-Commentary Source Line of code).

4.1.2 Platform and Software Compatibility

The rl software is written in Unix Korn shell scripting language [Bolsky and Korn 89]. The software was developed on an Amiga with the AT&T V. 4 Unix operating system and on a DEC with the Ultrix operating system. The rl software is dependent on the availability of the following software tools:
cflow

Cflow is available on most versions of the Unix operating system. This tool scans C source code files and produces a hierarchical listing of functions that are called within the files.

cscope

Cscope is available only on the AT&T Unix V. 4 operating system. Similar to cflow, cscope also scans C source files and produces listings and other information regarding the hierarchical structure of the system. The cscope tool is used in addition to cflow because it is capable of producing a hierarchical calling chart that mimics the multicall graph (see Section 3.2). Cscope is necessary only for producing metrics for the multicall graph abstraction; all other functions of rl are valid on any Unix platform.

ccount

Developed by Frakes, Fox and Nejme (Frakes et al 91), ccount counts the number of lines of non-commentary source code in C files.

The rl software has been bundled into a package containing the rl program, the rl manual page, a set of test C programs, the ccount software, and a README file. It is distributed on public domain and is available through anonymous ftp from ftp.vt.edu, in the directory pub/reuse.
4.2 rl Enhancements

This project includes several extensions to the rl software. Using command-line options, the user may specify different combinations of flags and arguments to control the behavior of the rl program. By default, the program reports the reuse level, reuse frequency and reuse complexity weighting for the given set of C files.

To clarify the myriad functions incorporated into rl, I have designed a Reuse Metric / Abstraction Matrix, shown in Table 15. Each column is labeled with a high level / low level abstraction combination. Each row is a metric calculated by rl. Each cell in the matrix provides a definition of the metric for the associated high level and low level abstraction. A cell value of 'NI' means that the metric/abstraction is Not Implemented in the rl program. The abbreviation NCSL means Non-Commentary Source Lines of code.
Table 15: Reuse Metric / Abstraction Matrix

<table>
<thead>
<tr>
<th>REUSE LEVEL</th>
<th>System Function</th>
<th>File Function</th>
<th>Function NCSL</th>
<th>System NCSL</th>
<th>File NCSL</th>
<th>Function SLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internal</td>
<td>Number of reused internal functions in all C files divided by the total number of functions used</td>
<td>Number of functions defined in the file and reused in the file divided by the total number of functions used in the file</td>
<td>Number of functions defined in the system and reused in the function divided by the total number of functions used by the function</td>
<td>Number of reused NCSL in all C files divided by the total number of NCSL in all files</td>
<td>Number of reused NCSL in the C file divided by the total number of NCSL in the file</td>
<td>Number of reused NCSL in each function divided by the total number of NCSL in each function</td>
</tr>
<tr>
<td>External</td>
<td>Number of reused external functions in all C files divided by the total number of functions used</td>
<td>Number of functions not defined in the file but reused in the file divided by the total number of tns used in the file</td>
<td>Number of tns not defined in the system but reused in the ttn divided by the total number of tns used by the ttn</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>REUSE FREQUENCY</th>
<th>System Function</th>
<th>File Function</th>
<th>Function NCSL</th>
<th>System NCSL</th>
<th>File NCSL</th>
<th>Function SLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internal</td>
<td>Number of static calls to reused internal functions divided by the total number of calls in all C files</td>
<td>Number of calls to functions defined in the file and reused in the file divided by the total number of calls in the file</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>External</td>
<td>Number of static calls to reused external functions divided by the total number of calls in all C files</td>
<td>Number of calls to tns not defined in the file but reused in the file divided by the total number of calls in the file</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>COMPLEXITY WEIGHTING</th>
<th>System Function</th>
<th>File Function</th>
<th>Function NCSL</th>
<th>System NCSL</th>
<th>File NCSL</th>
<th>Function SLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internal</td>
<td>Sum of NCSL of internal reused functions divided by the NCSL of all C files</td>
<td>Sum of NCSL of tns defined and reused in the file divided by the total NCSL in the file</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>
The following sections discuss the implementation of each rl enhancement.

4.2.1 Definition of Reuse Frequency

In addition to reporting the reuse level for C code, rl reports the reuse frequency. As discussed in Section 3.3.1, reuse frequency is based on the number of static references to reused components. For C code, the internal reuse frequency is the number of references to reused internal functions divided by the total number of references in the system. The external reuse frequency is the number of references to reused external functions divided by the total number of references in the system.

Recall that the definition of uses varies for different abstractions. Using a call graph abstraction for C code, function A uses function B one time if A calls B one or more times; this uses relation is 1 reference. Using a multicall graph abstraction, function A uses function B each time A calls B. Each call is then a reference. The rl program differentiates the reuse frequency for a standard call graph abstraction and a multicall graph abstraction accordingly. Rl does not count dynamic references or recursive calls.

Figure 3 shows sample rl output for reuse frequency.
4.2.2 Complexity Weighting

RL computes the reuse complexity weighting for internal functions. (See Section 3.3.2 for general information regarding the complexity weighting metric.) The ccound tool, discussed in Section 3.3.2, is used to count the number of NCSL in C source code. If ccound in unavailable, the complexity weighting will not be calculated. Ccound is dependent upon a delimeter between each function in the source code. This delimeter may be specified to rl using the -d parameter on invocation:

```
rl -d "/**new func**/" myprog.c
```

If the delimeter is not given as an argument to rl, the environment variable FDELIM is used as the delimeter. If FDELIM is also unvalued, rl will display a message and the complexity weighting will not be computed:

```
No file delimeter given, complexity weighting will not be computed.
```

Given a delimeter, rl calculates the reuse complexity weighting as the number of NCSL in the higher level component (system or file) divided by the sum of the NCSL for each reused internal function. Referring to Table 15, the complexity weighting is valid only
for System/Function and File/Function abstractions. Figure 4 shows sample rl output for the reuse complexity weighting:

```
Complexity Weighting
-----------------------
Total Non-Commentary Source Lines of code: 46
Total Non-Commentary Source Lines of reused code: 25
Complexity Weighting based on size: 0.543478
```

*Figure 4: rl Reuse Complexity Weighting Output*

### 4.2.3 Specification of internal threshold level and external threshold level

The user may specify values for internal threshold level (ITL) and external threshold level (ETL). Section 3.2 defines ITL as "the maximum number of uses of an internal item that can occur before reuse occurs." ETL is "the maximum number of uses of an external item that can occur before reuse occurs." The default value for ITL is 1 and ETL is 0. For a high level component of system and a low level component of function, this means that an internal function must be used at least two times for it to be "reused." An external function is "reused" when used only once.

The ITL and ETL may be given as command line arguments to rl:

```
rl -itl 2 -etl 1 myprog.c myutils.c
```

Rl always displays the ITL and ETL at the beginning of the output report:

```
Internal Threshold Level = 2
External Threshold Level = 1
```
RL uses the ITL and ETL for all calculations of reuse, for any abstraction. If ITL = 2, then a source line of code must occur three times for it to qualify as reused in the calculation of internal reuse level for low level component of NCSL.

4.2.4 Option to use multical graph abstraction

Given the –multical flag as a command line argument, rl will use a multical graph abstraction to calculate the reuse metrics. RL needs the cscope tool for this function. Cscope is available only on the AT&T Unix platform. When using a multical graph abstraction rl displays a message at the beginning of the output report:

   Using a multical graph abstraction with cscope.

If cscope is unavailable when the multical option is specified, rl displays an error and ends execution:

   rl error: Cannot use a multical abstraction.
   The cscope tool is unavailable; valid on AT&T Unix only.

Section 3.2 of this paper discusses the multical graph abstraction for generic higher level items and lower level items. RL includes a multical option for each metric except for a higher level component of function; the Function / Function abstraction is essentially always calculated using a multical abstraction since reuse is contained within a single function.
4.2.5 High level and low level abstractions

The original version of rl calculated the reuse level for functions within a C system. I have enhanced the software to calculate the reuse level metrics at various levels of abstraction. Each column in Table 15 represents a possible high level and low level abstraction combination. The table provides definitions of the metrics for each abstraction.

The default high level abstraction in rl is system. By using the \texttt{-h} flag the user may request a high level abstraction of file or function. For the system and file high level abstraction, the reuse level for a low level abstraction of function and NCSL are both calculated. For a high level abstraction of function, only the function low level abstraction is available at the current time.

4.2.6 Option to include main() in the function count

The final rl extension is an option to include the main() function in the count of internal functions. Specified on the command line, the \texttt{-main} flag causes rl to add 1 to the total count of internal functions when main() is one of those functions. By default, rl does not include main() in the count, a sensible approach because C allows only one definition of main() in a system and thus it is impossible to "reuse" the main() function.
4.3 Examples of use

The usage statement for rl is:

```
Usage: rl [-i num][-main][-multicall][-high system|file|function]
       [-d delim] <FILES>
```

Computes reuse metrics for the source code in the given file(s).

Parameters:
- `-i num`  num is the internal threshold level, default = 1
- `-e num`  num is the external threshold level, default = 0
- `-main`   includes main in the internal function count
- `-multicall` use a multicall graph abstraction
- `-high system|file|function`
  indicate system, file or function as the high level abstraction entity. Default is system.
- `-sysf filename`
  filename is the fully-qualified pathname of a file containing a list of system functions that should be excluded from reuse counts. If this parm is not given then all C functions will be included in reuse counts.
- `-d delim`  delim is a delimiter preceding each function definition in each C file, needed for computing the complexity weighting. If this parm is not given the environment variable FDELIM will be used. If neither are set then the complexity weighting will not be calculated.

`<FILES>` name of C source code files

The only required argument to rl is one or more names of C source files. Figure 5 is the report generated by rl for a C system composed of four files, `bv.c`, `hash.c`, `bvdriver.c`, and `hdriver.c`. The output resulted from invoking rl with no optional arguments:

```
rl hash.c bvdriver.c bv.c hdriver.c
```
No file delimiter given, complexity weighting will not be computed.
Internal Threshold Level = 1
External Threshold Level = 0

---

REUSE METRICS FOR SYSTEM
---

Reuse Level
-------------------
Internal Functions Reused: 9
External Functions Reused: 7
Total Functions Used: 30

Internal Reuse Level: 0.3000
External Reuse Level: 0.2333
Total Reuse Level: 0.5333

Reuse Frequency
-------------------
References to Reused Internal Functions: 50
References to Reused External Functions: 17
Total Number of References: 61

Internal Reuse Frequency: 0.6172
External Reuse Frequency: 0.2098
Total Reuse Frequency: 0.8271

Lines of Code Reuse Level
-------------------
LOC: 289
Reused LOC: 149
Reuse level for LOC: 0.5155

---

Figure 5: Sample rl output

Note that the default high level component is system; the high level component is displayed as the last word in the heading "REUSE METRICS FOR ...". Reuse levels for low level components of function and lines of code were calculated. No complexity weighting was computed because no function delimiter was defined.

Consider invoking rl with a high level component of file:

```
rl -high file hash.c bvdriver.c bv.c hdriver.c
```
The reuse metrics will be displayed for each file with appropriate headings such as:

---

REUSE METRICS FOR hash.c
---
4.3.1 Running rl from a makefile

A Unix makefile is often used to simplify the compilation and linking procedures for C programs. Since rl needs the names of all C source files to compute reuse metrics for a system, using the makefile will simplify running rl.

For example, the following lines can be added to a makefile:

```plaintext
RLARGS= 
RLDIR=/ul/cterry/RL
rl:
    $(RLDIR)/rl $(RLARGS) $(CSRC)
```

RLDIR is the directory path of the rl software. The CSRC variable is the list of C source files. The rl program can be invoked by typing:

```plaintext
make rl
```

Additional arguments may also be specified:

```plaintext
make rl RLARGS="-itl 2 -high file"
```

To modify the default behavior of rl using a makefile, assign the value for the RLARGS variable inside the makefile.
5 Testing rl

5.1 The Testing Procedure

All rl capabilities have been extensively tested. The program incorporates parameter edits and other error-prevention procedures that improve the robustness of the software. A set of seven very simple C programs were used as test cases for the initial testing phases of rl. The programs were designed to provide simple but comprehensive test cases.

The following is a regression test script for rl. It was adapted from a similar script on page 197 in [Frakes et al 91]. This script accepts two arguments. The first is the name of a file which contains the correct results for the test run. The second is all arguments that should be passed to the rl program.

```
# Shell script to test the rl program
#
# $1 - name of file containing correct results
# $2 - all parms (enclosed in quotes) to rl

CORRECT=$1

echo Testing rl
echo Comparing results on reuse metrics
echo to results in $CORRECT
```
```
echo

# run the rl program, capture stdout and stderr

echo "rl $2"
rl $2 > /tmp/tmp.r1out 2>&1

# test for differences in this output and correct output

diff /tmp/tmp.r1out $1 > /tmp/diffs.$CORRECT
if test "$?" = 1
then
    echo Error - Differences are:
cat /tmp/diffs.$CORRECT
else
    echo No errors
    rm /tmp/tmp.r1out
    rm /tmp/diffs.$CORRECT
fi
```

This script was used to test the rl program on each of the seven simple programs. The C test programs are listed in Appendix D. The following list summarizes the rl test cases. This list is not comprehensive; additional test cases provide better coverage of parameter edits and argument combinations.

1. rl myprog.c (correct, no options)
2. rl nofile.c (bogus filename)
3. rl myprog.c myutil.c (2 correct file names)
4. rl (no file names)
5. rl -itl 2 -etl 1 myprog.c (specify threshold levels)
6. rl -main myprog.c (-main argument)
7. rl -multicall myprog.c
   (use multicall graph abstraction)
8. rl -high file myprog.c myutil.c
   (high level component = file)
9. rl -high file -multicall
   myprog.c myutil.c
   (high level=file, with multicall abstraction)
10. rl -high function myprog.c
    myutil.c
    (high level component = function)
11. rl -d "/*%"
    (-d flag and FDELIM env. variable)
12. rl -high bogus
    (bogus high level component)

5.2 Testing Results

The rl software was run on 31 production C systems. The systems range in size from 192
NCSL to 1879 NCSL. Access to C software was limited for this project; it would be
beneficial to run further analysis on larger systems and systems with existing quality and
productivity statistics. The following data was accumulated using default rl options:
internal threshold level = 1, external threshold level = 2, standard call graph abstraction,
high level component = system and low level component = function. Table 16 shows
summary statistics for the reuse level metric.
Table 16: Summary Statistics for Reuse Level

<table>
<thead>
<tr>
<th></th>
<th>Internal Reuse Level</th>
<th>External Reuse Level</th>
<th>Total Reuse Level</th>
<th>NCSL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>0.0858</td>
<td>0.5522</td>
<td>0.6380</td>
<td>818.84</td>
</tr>
<tr>
<td>Median</td>
<td>0.0666</td>
<td>0.5263</td>
<td>0.6190</td>
<td>676</td>
</tr>
<tr>
<td>Max</td>
<td>0.3000</td>
<td>0.9814</td>
<td>0.9814</td>
<td>1879</td>
</tr>
<tr>
<td>Min</td>
<td>0</td>
<td>0.2333</td>
<td>0.4655</td>
<td>192</td>
</tr>
<tr>
<td>Standard Dev</td>
<td>0.0735</td>
<td>0.1474</td>
<td>0.1220</td>
<td>471.82</td>
</tr>
</tbody>
</table>

The average level of reuse is high, at 64%. This figure could be partially attributed to the design of the C programming language. The internal reuse level ranges from zero (no internal reuse) to 0.3 (approximately one-third of the internal functions were used more than once). Figure 6 shows a plot of internal reuse level vs. log NCSL.
The plot would indicate that as software increases in size, internal reuse also increases. On the other hand, as shown in Figure 7, external reuse decreases as software increases in size. Internal reuse has a positive correlation with log NCSL ($r=0.35$, $r^2=0.12$). External reuse has a negative correlation with log NCSL ($r=-0.57$, $r^2=0.32$). These results indicate that small programs reuse a small number of internal functions and a large number of external components. Conversely, larger programs reuse more internal components and fewer external components. These results are logical because large programs contain more code and thus more reusable components, and small programs must use external components to increase functionality.
Table 17 contains summary statistics for reuse frequency.

**Table 17: Summary Statistics for Reuse Frequency**

<table>
<thead>
<tr>
<th></th>
<th>Internal Reuse Frequency</th>
<th>External Reuse Frequency</th>
<th>Total Reuse Frequency</th>
<th>NCSL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>0.1308</td>
<td>0.6525</td>
<td>0.7833</td>
<td>818.84</td>
</tr>
<tr>
<td>Median</td>
<td>0.0983</td>
<td>0.6666</td>
<td>0.7794</td>
<td>676</td>
</tr>
<tr>
<td>Max</td>
<td>0.6172</td>
<td>0.9838</td>
<td>0.9838</td>
<td>1879</td>
</tr>
<tr>
<td>Min</td>
<td>0</td>
<td>0.2098</td>
<td>0.6363</td>
<td>192</td>
</tr>
<tr>
<td>Standard Dev</td>
<td>0.1261</td>
<td>0.1379</td>
<td>0.0725</td>
<td>471.82</td>
</tr>
</tbody>
</table>
As expected, because more calls to reused functions exist than reused functions themselves, the reuse frequency measurements are slightly above the reuse level statistics.

An additional study evaluates the effect of increasing the threshold level on the corresponding reuse level. Figure 8 maps the internal reuse level to the internal threshold level. This data was acquired by running rl on a single set of software two different times, once with an internal threshold level of 1 and once with an internal threshold level of 2. The graph indicates that the reuse level drops significantly when the threshold level was increased by 1. All but one test case has a reuse level of 0.075 or less with ITL=1.

*Figure 8: Internal reuse level vs. Internal threshold level*
Figure 9 shows an even more dramatic difference in external reuse level when the external threshold level is increased from 0 to 1. An ETL of 1 means that an external function must be used more than once before "reused."

![Diagram of external reuse level vs. external threshold level]

*Figure 9: External reuse level vs. External threshold level*

Further testing of rl to evaluate the reuse levels using a multical graph abstraction would be beneficial. Results for such tests are not included in this report due to a computer hardware failure.
6 Conclusions and Future Work

In this study, the reuse metrics proposed by Frakes were extended to include reuse frequency and a reuse complexity weighting. Threshold levels and a multicall graph abstraction were defined to allow variable definitions of "reuse." The metrics were formally defined in an algorithmic notation that allows clear understanding. The formal definition can be mapped to a specific domain and thus the reuse metrics can be defined for any software development artifact.

The rl reuse measurement software for C code has been greatly enhanced. It now reports the reuse level and reuse frequency metrics for several possibilities of high level and low level abstraction entities. Rl allows the user to specify values for internal and external threshold levels. It also computes a reuse complexity weighting, and supports calculation of the reuse metrics using a multicall graph abstraction.

Runs of the rl software conclude that internal reuse increases as programs increase in size while external reuse decreases as program increase in size. Also, a difference of one in the internal or external threshold level causes dramatic decreases in the amount of reuse.

Much future work remains to be done. For clearer empirical results, the reuse metrics could be calculated for systems that have prior quality and productivity measurements. Correlations could then be obtained providing empirical evidence of the effect of software reuse on software quality and productivity.
The abstract definitions of the reuse metrics could be mapped to the object-oriented domain to reflect the syntax of an object-oriented language. Rl or similar software could be written to calculate reuse measurements for an object-oriented language such as C++. Valuable results could compare the amount of reuse in procedural source code to the amount of reuse in object-oriented code.

The rl program could be enhanced to compute dynamic reuse metrics, calculated after source code bindings have been resolved. Recursion and dynamic references would then be included in the reuse equation, as they should be for more precise results.

As currently implemented, the rl software is dependent upon several other tools to parse the source code. An improved implementation might incorporate the parsing procedure and generate results more efficiently.

To improve accessibility, the functions of a reuse measurement tool such as rl could easily be integrated into a CASE environment. Users could then graphically request reuse statistics for a given software component.

Software reuse is predicted to become more common and more obtainable in the future of software development. Measurement tools such as rl will be an essential part of the reuse program.
References


Appendix A: Definition of Reuse Metrics in Set Notation

This appendix contains a formal definition of the reuse level and reuse frequency metrics. The definitions are constructed in formal set notation. Section A.1 contains an abstract definition for unknown high level and low level entities. Section A.2 demonstrates how the general notation can be applied to an actual domain to define a reuse measurement model for the C programming language.

A.1 Abstract Formal Reuse Metric Definitions

Assumptions

Parts-based reuse assumes that a system is divided into parts, or components, which relate to each other in some way. Higher-level components are composed of, or use, lower-level components. A component which is internal to a system was developed for the system. An external component was not developed for the system but may be used within the system.

Such a system can be modeled with nodes and relationships between nodes, as in a call graph abstraction. In Figure 3, each node represents some object in the development life cycle of system S. When a node (A1) uses another node (A2) then the relationship from A1 to A2 is a uses relation, or a reference. A reference is depicted as a directional arc between nodes.
The system in Figure 3 does not show whether a node is used by another node more than once. By definition, the reference relation is boolean: either A1 uses A2 or it does not. An alternative representation is a multical call graph abstraction where a node may be used several times by another node. In Figure 4, each arc is labeled with a digit which indicates the number of uses. A1 uses A2 twice and A4 uses A5 three times.

While more definitions are possible, two basic reuse metrics are defined:

1) The amount of reuse may be defined in terms of the number of nodes which are used more than a given number of times within the system. The resulting metric is reuse level [Frakes 90]. Reuse level is defined as the number of nodes which are reused divided by the total number of nodes in the system.

2) The amount of reuse may be defined in terms of the number of references to nodes.

The resulting metric is reuse frequency. Reuse frequency is defined as the
number of references to reused nodes divided by the total number of references in the system.

Reuse level and reuse frequency may both be decomposed into internal and external measurements.

For the purpose of a formal definition, the following assumptions are made:

Let $S$ be a system.

Let $N$ be the set of all nodes developed for $S$ (internal nodes).

Let $R$ be the set of all references within $S$ (to internal and external nodes).

Let $ITL$ be the internal threshold level, defined to be the maximum number of uses of an internal node that can occur before reuse occurs. The default value is 1.

Let $ETL$ be the external threshold level, defined to be the maximum number of uses of an external node that can occur before reuse occurs. The default value is 0.

**Definitions**

$T =$ total # of nodes in $S$

$IU =$ # internal nodes in $S$ used $> ITL$

$IM =$ # internal nodes in $S$ used $> ITL$
with a multical call graph abstraction

$EU =$ # external nodes used $> ETL$

$TF =$ total # of references in $S$

$IUF =$ # references in $S$ to internal nodes used $> ITL$

$IMF =$ # references in $S$ to internal nodes used $> ITL$ with a multical graph abstraction

$EUF =$ # references in $S$ to external nodes used $> ETL$
EM = \# external nodes used > ETL with a multicall graph abstraction
IMF = \# references in S to external nodes used > ETL with a multicall graph abstraction

T is equivalent to the formal relation \{ x \mid \text{used}_\text{cnt}(x) \geq 1 \} \}

TF is equivalent to the formal relation \{ R \}

The function used\_cnt(a) is the number of times the component a is used in S using a standard call graph abstraction.

The function multicall\_used\_cnt(a) is the number of times the component a is used in S using a multicall graph abstraction.

The function ref\_cnt\_to\_node(a) is the number of references to a within S using a standard call graph abstraction.

The function multicall\_ref\_cnt\_to\_node(a) is the number of references to a within S using a multicall graph abstraction.

**Internal Reuse Level**

\[
i\_\text{reuse\_level} = IU / T
\]

where \( IU = \{ (x \in N) \mid \text{used\_cnt}(x) > ITL \} \)

\[
i\_\text{reuse\_level\_multicall} = IM / T
\]

where \( IM = \{ (x \in N) \mid \text{multicall\_used\_cnt}(x) > ITL \} \)

**External Reuse Level**

\[
e\_\text{reuse\_level} = EU / T
\]

where \( EU = \{ (x \notin N) \mid \text{used\_cnt}(x) > ETL \} \)
\[
e_{\text{reuse\_level\_multicall}} = \frac{EM}{T}
\]

\[
\text{where } EM = \{ (x \notin N) \mid \text{multicall\_used\_cnt}(x) > ETL \} \}
\]

**Internal Reuse Frequency**

\[
i_{\text{reuse\_frequency}} = \frac{\text{IUF}}{TF}
\]

\[
\text{where } \text{IUF} = \{ (d \in R) \mid \text{ref\_cnt\_to\_node}(d) > ITL \land \\
\text{ref\_to\_internal\_node}(d) \}
\]

\[
i_{\text{reuse\_frequency\_multicall}} = \frac{\text{IMF}}{TF}
\]

\[
\text{where } \text{IMF} = \{ (d \in R) \mid \text{multicall\_ref\_cnt\_to\_node}(d) > ITL) \\
\land \text{ref\_to\_internal\_node}(d) \}
\]

**External Reuse Frequency**

\[
e_{\text{reuse\_frequency}} = \frac{\text{EUF}}{TF}
\]

\[
\text{where } \text{EUF} = \{ (d \in R) \mid \text{ref\_cnt}(d) > ETL \land \\
\text{ref\_to\_external\_node}(d) \}
\]

\[
e_{\text{reuse\_frequency\_multicall}} = \frac{\text{EMF}}{TF}
\]

\[
\text{where } \text{EMF} = \{ (d \in R) \mid \text{multicall\_ref\_cnt\_to\_node}(d) > ETL) \\
\land \text{ref\_to\_external\_node}(d) \}
\]

**Total Reuse Measurements**

\[
total_{\text{reuse\_level}} = i_{\text{reuse\_level}} + e_{\text{reuse\_level}}
\]

\[
total_{\text{reuse\_level\_multicall}} = i_{\text{reuse\_level\_multicall}} + e_{\text{reuse\_level}}
\]

\[
total_{\text{reuse\_frequency}} = i_{\text{reuse\_frequency}} + e_{\text{reuse\_frequency}}
\]
total_reuse_frequency_multicall = i_reuse_frequency_multicall + e_reuse_frequency

**Complexity (size) Weighting**

\[
W_S = \frac{\sum_{i=1}^{n} \text{(size (internal_reused_nodes(ITL)))}}{\sum_{i=1}^{n} \text{(size (all_nodes(S)))}}
\]

where size is based on NCSL (Non-Commentary Source Lines of code).

### A.2 Formal Reuse Metric Definitions for the C Language

This section demonstrates how the formal model can be instantiated for C. The high level abstraction for this definition is a C system. The low level abstraction is a function. The following assumptions are made:

Let S be a software system written in the C programming language. Functions that were developed for S are called internal functions and functions that were not developed for S are called external functions.

Let F be the set of all internal functions.

Let C be the set of all references within S to internal and external functions.

Let ITL be the internal threshold level, defined to be the maximum number of references to an internal function that can occur before reuse occurs.

Let ETL be the external threshold level, defined to be the maximum number of references to an external function that can occur before reuse occurs.
Definitions

T=total # of functions  TF=total # of references
IU=# internal functions referenced > ITL  IUF=# references to internal functions
    used > ITL
IM=# internal functions referenced > ITL  IMF=# references to internal functions
    with a multical graph abstraction  used > ITL with a multical graph abstraction
EU=# external functions referenced > ETL  EUF=# references to external functions
EM=# external functions referenced > ETL  EMF=# references to external functions
    with a multical graph abstraction  used > ETL with a multical graph abstraction

T is equivalent to the formal relation \( \{ f \mid \text{used}_\text{cnt}(f) \geq 1 \} \)

TF is equivalent to the formal relation \( \{ C \} \)

The function \( \text{used}_\text{cnt}(f) \) is the number of times function \( f \) is used in \( S \) using a standard call graph abstraction.

The function \( \text{multical}_\text{used}_\text{cnt}(f) \) is the number of times the function \( f \) is used in \( S \) using a multical graph abstraction.

The function \( \text{ref}_\text{cnt}(c) \) is the number of references to function \( c \) using a standard call graph abstraction.

The function \( \text{multical}_\text{ref}_\text{cnt}(c) \) is the number of references to function \( c \) using a multical graph abstraction.
Internal Reuse Level

\[ i_{\text{reuse\_level}} = \frac{IU}{T} \]
where \( IU = \{ (f \in F) \mid \text{used\_cnt}(f) > \text{ITL} \} \)

\[ i_{\text{reuse\_level\_multicall}} = \frac{IM}{T} \]
where \( IM = \{ (f \in F) \mid \text{multicall\_used\_cnt}(f) > \text{ITL} \} \)

External Reuse Level

\[ e_{\text{reuse\_level}} = \frac{EU}{T} \]
where \( EU = \{ (f \notin F) \mid \text{used\_cnt}(f) > \text{ETL} \} \)

\[ e_{\text{reuse\_level\_multicall}} = \frac{EM}{T} \]
where \( EM = \{ (f \notin F) \mid \text{multicall\_used\_cnt}(f) > \text{ETL} \} \)

Internal Reuse Frequency

\[ i_{\text{reuse\_frequency}} = \frac{IUF}{TF} \]
where \( IUF = \)
\[ \{ (c \in C) \mid \text{ref\_cnt}(c) > \text{ITL} \land \text{ref\_to\_internal\_ftn}(c) \} \]

\[ i_{\text{reuse\_frequency\_multicall}} = \frac{IMF}{TF} \]
where \( IMF = \{ (c \in C) \mid \text{multicall\_ref\_cnt}(c) > \text{ITL} \land \text{ref\_to\_internal\_ftn}(c) \} \)
External Reuse Frequency

\[ e\_\text{reuse\_frequency} = \frac{\text{EUF}}{\text{TF}} \]

where \( \text{EUF} = \)

\[ | \{(c \in C) \mid \text{ref\_cnt}(c) > \text{ETL} \land \text{ref\_to\_external\_ftn}(c)\}| \]

\[ e\_\text{reuse\_frequency\_multicall} = \frac{\text{EMF}}{\text{TF}} \]

where \( \text{EMF} = | \{(c \in C) \mid \text{multicall\_ref\_cnt}(c) > \text{ETL}\} \land \text{ref\_to\_external\_ftn}(c)\} | \]

Total Reuse Measurements

\[ \text{total\_reuse\_level} = i\_\text{reuse\_level} + e\_\text{reuse\_level} \]

\[ \text{total\_reuse\_level\_multicall} = i\_\text{reuse\_level\_multicall} + e\_\text{reuse\_level} \]

\[ \text{total\_reuse\_frequency} = i\_\text{reuse\_frequency} + e\_\text{reuse\_frequency} \]

\[ \text{total\_reuse\_frequency\_multicall} = i\_\text{reuse\_frequency\_multicall} + e\_\text{reuse\_frequency} \]

Complexity (size) Weighting

\[ W_s = \sum_{i=1}^{n} \frac{\text{size}(\text{internal\_reused\_ftns}(S,\text{ITL}))}{\text{size}(\text{all\_ftns}(S))} \]

where size is based on NCSL (Non-Commentary Source Lines of code).
Appendix B: The rl Software
#!/usr/bin/ksh


# Environment: Will use the following environment variables if defined:
#   ITL - internal threshold level, default=1
#   ETL - external threshold level, default=0
#   FDElim - file delimiter for ccount

# Distribution: Not to be distributed without permission from B. Frakes

# Author: Carol Terry, Va Tech (703) 698-6020
#         Bill Frakes, Va Tech (703) 698-6020
#         Chris Fox

# History:
# 6-22-90 Written by B. Frakes
# 1-30-92 Rewritten by C. Fox
# 6-20-93 Rewritten by C. Terry

# Reference: W. Frakes, "An Empirical Program for Software Reuse"
# 3rd Workshop on Tools and Methods for Reuse
# Syracuse, NY 1990
# C. Terry, "Analysis and Implementation of Software Reuse Measurement"
# Masters Project and Report, VPI & SU Northern Virginia
# Falls Church, VA 1993

# Notes: Run rl out of your makefile using your source list as input.

TMPDIR=/tmp    # directory to store temporary files
HLQ=/u/cterry/carol/RL    # high level qualifier (path) for the rl program
CCNT_DIR=/u/cterry/carol/RL    # path for the ccount program
#HLQ=/home/cterry/rl    # path for rl on amiga
#CCNT_DIR=/home/cterry/rl    # path for the ccount program on amiga

# --------
# Usage
# --------

Usage() {
    echo "Computes reuse metrics for the source code in the given file(s)"
    echo "Parameters:"
echo "-lt1 num num is the internal threshold level, default = 1"
echo "-et1 num num is the external threshold level, default = 0"
cho "-main include main in the internal function count"
echo "multcall use a multical call graph abstraction so that, for exa

   a function can use another function more than once

echo "-high [system][file][function]"
chosw system, file or function as the high lev
cho "-sysf filename"
cho filename is the fully-qualified pathname of a file
cho "contain a list of system functions that should
chosw excluded from reuse counts. If this parm is not g
chosw then all C functions will be included in reuse cou
cho "-d delim delin is a delimiter preceding each function defin
chosw in each C file, needed for computing the complexit
cho "-y size enten"
cho "-hen"
chosw <FILES>
exite 1
}

# *------------------------------------------------------------------
# *------------------------------------------------------------------
# Parmedit - edit all input parms, if errors print Usage stmt and exit
# *------------------------------------------------------------------
# *------------------------------------------------------------------

Parmedit() {
  if [ $# -eq 0 ]
    then
      Usage
    fi

  # search for valid parameters. Store the names of the c files in the CF
  # ILE array.
  let CNT=0
  while [ $# -ne 0 ]
    do
      case $1 in
        -lt1) shift
           ITL=$1
          ;;
        -et1) shift
           ETL=$1
          ;;
        -main) COUNTMAIN=yes
           echo "main is included in the count of Internal Functions "
          ;;
        -multical) if [! -n `whence cscope`]}

95
then
  MULTICALL=yes
  echo "Using a multical abstraction with cscope."
else
  echo "rl error: Cannot use a multical abstraction."
  echo "The cscope tool is unavailable; valid on AT&T Unix only."
Usage
fi

;: -sysf) shift
  SYSFTN_FILE=$1
  ::
-high) shift
  ABSTRACTION=$1
  ::
-d) shift
  MY_FDELM=$1
  ::
)*CFILE[$CNT]=$1
  let CNT=CNT+1
  ::
esac
shift
done

#edit high level abstraction
if [[ -n $ABSTRACTION ]]
then
  case $ABSTRACTION in
  *system) ;;
  *file) ;;
  *function) ;;
  *) echo rl error: high level abstraction $ABSTRACTION is not valid
Usage
esac
fi

#check for existence of function delimiter for ccount
if [[ -z $MY_FDELM ]]
then
  if [[ -z $FDELM ]]
  then
    NO_FDELM=true
    echo No file delimiter given, complexity weighting will not be computed.
  fi
fi

#edit source file names
if [[ -z $CFILE ]]
then
  echo rl error: no source filenames given
Usage
fi
for filen in *${CFILE[*]}
do
  if [[ ! -f $filen ]]
  then
    echo rl error: no such file: $filen
Usage
  fi
done

if [[ -n $SYSFTN_FILE ]]

96
then
  if [[ ! -f $SYSFTN_FILE ]]
    then
      echo rl error: no such file: $SYSFTN_FILE
    fi
fi

#edit default directories
if [[ ! -d $TMPDIR ]]
  then
    echo rl error: no such directory: $TMPDIR
    Usage
fi
if [[ ! -d $HLQ ]]
  then
    echo rl error: no such directory: $HLQ
    Usage
fi
if [[ ! -d $CCNT_DIR ]]
  then
    echo rl error: no such directory: $CCNT_DIR
    Usage
fi

#set defaults for ITL and ETL if necessary
if [[ -z $ITL ]]
  then
    ITL=1
fi
if [[ -z $ETL ]]
  then
    ETL=0
fi

# Reuse_Level_Metrics - generates:
# internal_reuses = # internal functions reused
# external_reuses = # external functions reused
# total_uses = # functions used
#-------------------------------------
#-------------------------------------

Reuse_Level_Metrics()
{
  #------------------------------------------------------------------------

  # Catalog the definitions of internal functions by searching for matched
  # parentheses. Keep only the name and definition line number. Then
  # extract uses of these defined functions from the cflow output or from
  # cscope.  Count how often each internal function is called.

  grep "\(\)" $TMPDIR/cflow.file | awk '{print $2}' > $TMPDIR/intern_defs

  if [[ -z $MULTICALL ]]
    then
      for function in `cat $TMPDIR/intern_defs`
        do
          ...
if [[ $function != "main:" ]]
    then grep "$function" $TMPDIR/cflow.file | awk '{print $2}'
fi
done > $TMPDIR/intern_calls

# NOTE: for AT&T Unix only use cscope;
# Allow option of defining "reuse" as multiple calls within a single function.
# remove the colon from the function name
for function in sed "s/://" $TMPDIR/intern_defs do
cscope -L -3 $function $(CFILE[*]) | awk '{print $4}' | sed "s/\(.*\\)/" done > $TMPDIR/intern_calls
fi

# Count the internal functions used at least once and more than ITL times,
# where ITL is set to 1 if undefined.
sort $TMPDIR/intern_calls | uniq -c > $TMPDIR/intern_counts

# Add 1 for main if indicated by the -m flag. The number of internal reuses
# is the number of components that are used more than ITL times.
intern_reuses=`cat $TMPDIR/intern_counts | wc -l`
if [[ -n $COUNTMAIN ]]
    then intern_reuses=`echo $intern_reuses +1 | dc`
fi

# Write to $HLQ/reused_nodes an ordered list of reused functions
awk '{if ($1$I) print $0}' $ITL $TMPDIR/intern_counts > $HLQ/reused_nodes

# Count the external functions called at least once by searching for matched
# empty angle brackets. Keep only the function name, then sort and unique
# the result, and count the distinct functions.
grep "<=" $TMPDIR/cflow.file | awk '{print $2}' | sort > $TMPDIR/externdefs

# use cscope to get a multicall abstraction. Do not count any function
# defined in the $SYSFTN_FILE, specified by the -sysf parm.
# $TMPDIR/externdefs
if [[ -z $ETL ]]
    then ETL=0
fi

# use cscope to get a multicall abstraction. Do not count any function
# defined in the $SYSFTN_FILE, specified by the -sysf parm.
if [[ -z $MULTICALL ]]
    then for function in `cat $TMPDIR/extern_defs`
        do if [[ -n $SYSFTN_FILE ]]
            then if [[ -z "grep $function $SYSFTN_FILE" ]]
                then grep "$function" $TMPDIR/cflow.file | awk '{print $2}' > $TMPDIR/extern_calls
                fi
            else grep "$function" $TMPDIR/cflow.file | awk '{print $2}' > $TMPD
DIR/extern_calls
  fi
  done
else
  for function in `sed "s/::/\" $TMPDIR/extern_defs``
    do
      if [[ -n $SYSPTN_FILE ]]
        then
          if [[ -z `grep $function $SYSPTN_FILE` ]]
            then
cscope -L -3 $function ${CFILE[*]} | awk '{print $4}' | sed "s/\{.*\}/" > $TMPDIR/extern_calls
            fi
          else
cscope -L -3 $function ${CFILE[*]} | awk '{print $4}' | sed "s/\{.*\}/" > $TMPDIR/extern_calls
          fi
    done
fi

# Count the external functions used at least once and more than ETL times,
# where ETL is set to 0 if undefined.
sort $TMPDIR/extern_calls | uniq -c > $TMPDIR/extern_counts
extern_uses= cat $TMPDIR/extern_counts | wc -l

# Write to reused exnodes an ordered list of reused external functions
awk '{if ($1>K) print 0} K=$ETL $TMPDIR/extern_counts > $TMPDIR/reused_exnodes
extern_reuses=`wc -l $TMPDIR/reused_exnodes | awk '{print $1}'`

#

# Reuse_Frequency Metrics - generates:
# internRefs = # references to reused internal functions
# externRefs = # references to reused external functions
# totalRefs = total # references
#
# Reuse_Frequency_Metrics()
#
# Compute the internal reuse frequency by counting number of references to each
# component. totalInternRefs is the total number of references to internal components.
# internRefs is the number of references above ITL to internal components.

totalInternRefs=0
for cnts in `awk '{print $1}' $TMPDIR/intern_counts`
do
totalInternRefs=`echo $cnts $totalInternRefs "+p" | dc`
done
awk '{if ($1>ITL) print $1}' K=$ITL $TMPDIR/intern_counts > $TMPDIR/internRefs_counts
internRefs=0
for cnts in `cat $TMPDIR/internRefs_counts`

do
    intern_refs=`echo $cnts $intern_refs "+p" | dc`
do
#
# Compute the external reuse frequency. Must first compile a list of all
# calls to external components.

    total_EXTERN_REFS=0
    for cnts in `awk '{print $1}' $TMPDIR/extern_counts`
do
        total_EXTERN_REFS=`echo $cnts $total_EXTERN_REFS "+p" | dc`
done

    awk '{if ($1>K) print $1}' K=$ETL $TMPDIR/extern_counts > $TMPDIR/extern.refs_counts
    extern.refs_counts=0
    for cnts in `cat $TMPDIR/extern.refs_counts`
do
        extern.refs_counts=`echo $cnts $extern.refs_counts "+p" | dc`
done
#
# --- ---------------------------
# --- ---------------------------
# ---
# Reuse Lines Metrics
#
# callgraph abstraction = # lines reused / # unique lines
# multicalcgraph abstraction = # occurrences of reused lines / total # lines
# Generates:
#    T_LINES = total number of lines in the high lev component
#    T_LINES_REUSED = number of reused lines
#
# ---------------------------
# ---------------------------
# ---------------------------

Reuse_Lines_Metrics()
#
# Compute internal reuse level for lines of code rather than functions as
# the lower level abstraction.

    fnames=$*
    for filen in $fnames
do
        awk '/\*/\*// {next} {print $0}' $filen # extract commentary lines
    done
    sort | uniq -c | awk '{if ($2) print $0}' > $TMPDIR/all_lines
    awk '{if ($1 > K) print $0}' K=$ITL $TMPDIR/all_lines > $TMPDIR/intern_lines
    if [[ -z $MULTICALL ]]
    then
        T_LINES=`wc -l $TMPDIR/all_lines` | awk '{print $1}' ` # number of unique lines
        T_LINES_REUSED=`wc -l $TMPDIR/intern_lines` | awk '{print $1}'`
    else
        T_LINES=0
        T_LINES_REUSED=0
        for aline in $TMPDIR/all_lines`
do
        T_LINES=`echo $aline $T_LINES "+p" | dc`

100
done
  for aine in `awk '{print $1}' $TMPDIR/intern_lines`
do
    T_LINES_REUSED=`echo $aine $T_LINES_REUSED "-p" | dc`
done
fi

Function_Reuse_Level_Metrics()
{
  echo Reuse Metrics for high level component = function
echo
cflow ${CFILE[*]} > $TMPDIR/cflow.file
grep "\(" $TMPDIR/cflow.file | awk '{print $2}' > $TMPDIR/intern_defs
let fnms=0
for fname in `sed "s/://" $TMPDIR/intern_defs`
do
  # this cscope call lists internal AND external functions called by
  # $fname; need to differentiate to generate irl and erl
  cscope -l -2 $fname ${CFILE[*]} | awk '{print $2}' > $TMPDIR/intern_callbys

  # sort and count functions called
  sort $TMPDIR/intern_callbys | uniq -c > $TMPDIR/intern_counts
  func Uses= wc -l $TMPDIR/intern_counts | awk '{print $1}'

  # count number of functions called > ITL times
  awk '{if ($1 > K) print $0}' K=$ITL $TMPDIR/intern_counts > $TMPDIR/$reused_nodes
  func_reuses=`wc -l $TMPDIR/reused_nodes | awk '{print $1}'`

  # output results
  echo ============
  echo REUSE METRICS FOR $fname
  echo ============
  echo Functions Reused: $func_reuses
  echo Total Functions Used: $func Uses
  echo $func_reuses $func Uses | awk '{printf("Reuse Level: %f\n",$1/$2)}'

101
echo
done

---

# Complexity_Weighting - valid only for high level component = SYSTEM or FILE
# low level component = FUNCTION
# Generates: 
#   T_NCSL   = total noncommentary source lines in given high level component
#   T_NCSL_REUSED = # noncommentary source lines in internal reused functions
# -------------------------------
Complexity_Weighting()
{
  fnames=$*
  
  # count NCSL in all C files
  if [ ![ -n $MY_FDELiM ]]
    then
      $CCNT_DIR/ccount $fnames -d $MY_FDELiM -t > $TMPDIR/ccount.out
    else
      $CCNT_DIR/ccount $fnames -t > $TMPDIR/ccount.out
  fi
  
  if [ ![ -f $TMPDIR/tcnts ]]
    then
      rm $TMPDIR/tcnts
    fi
  grep "total".$TMPDIR/ccount.out | awk '{print $3}' > $TMPDIR/tcnts
  for tcnt in cat $TMPDIR/tcnts
    do
    T_NCSL=`echo $tcnt $T_NCSL "+p" | dc`
  done
  
  # intern_ncs1 is a list of reused functions
  for function in 'awk '{if ($1 > K) print $2}' K=$ITL $TMPDIR/intern_counts | sed "s/:/ /"
    do
    grep $function $TMPDIR/ccount.out | awk '{print $3 " " $1}'
    done > $TMPDIR/intern_ncs1
  
  # sum the NCSL in all reused functions
  T_NCSL_REUSED=0
  for fnncsl in 'awk '{print $1}' $TMPDIR/intern_ncs1`
    do
    T_NCSL_REUSED=`echo $fnncsl $T_NCSL_REUSED "+p" | dc`
  done
}

---

# Output_Report - write pretty report to stdout
---
Output_Report()
{
    echo "Reuse Level"
    echo $intern_reuses | awk '{ printf("Internal Functions Reused: %d\n", $1) }'
    echo $extern_reuses | awk '{ printf("External Functions Reused: %d\n", $1) }'
    echo $totalUses | awk '{ printf("Total Functions Used: %d\n", $1) }'
    echo $intern_reuses $totalUses | awk '{ printf("Internal Reuse Level: \%f\n", ($1/$2)) }'
    echo $extern_reuses $totalUses | awk '{ printf("External Reuse Level: \%f\n", ($1/$2)) }'
    echo $intern_reuses $total_reuses $extern_reuses | awk '{ printf("Total Reuse Level: \%f\n", ($1/$2)+(3/$2)) }'
    echo "Reuse Frequency"
    echo $intern_refs | awk '{ printf("References to Internal Functions: %d\n", $1) }'
    echo $extern_refs | awk '{ printf("References to External Functions: %d\n", $1) }'
    echo $total_refs | awk '{ printf("Total Number of References: %d\n", $1) }'
    echo $intern_refs $total.refs | awk '{ printf("Internal Reuse Frequency: %f\n", ($1/$2)) }'
    echo $extern.refs $total.refs | awk '{ printf("External Reuse Frequency: %f\n", ($1/$2)) }'
    echo $intern.refs $total.refs $extern.refs | awk '{ printf("Total Reuse Frequency: %f\n", ($1/$2)+(3/$2)) }'
    echo "Lines of Code Reuse Level"
    echo "NCSL (Non-Commentary Source Lines): " $T_LINES
    echo "Reused NCSL: " $T_LINES_REUSED
    echo $T_LINES_REUSED "$T_LINES" | awk '{ printf("Reuse level for NCSL: %f\n", $1/$2) }'
    if [[ -z $NO_FDELIM ]]
    then
        echo "Complexity Weighting"
        echo $T_NCSL | awk '{ printf("Total Non-Commentary Source Lines of code: %d\n", $1) }'
        echo $T_NCSL_REUSED | awk '{ printf("Total Non-Commentary Source Lines of reused code: %d\n", $1) }'
        echo $T_NCSL_REUSED "$T_NCSL" | awk '{ printf("Complexity Weighting based on size: %f\n", $1/$2) }'
    fi
}

# Output_Stats - write stats to files in stats directory, brief output to stdout
# this routine is for testing purposes only
Output_Sstats()
{
    echo $intern_reuses $total_uses | awk '{printf("%f\n", $1/$2)}' >> $HLQ/stats/irl
    echo $extern_reuses $total_uses | awk '{printf("%f\n", $1/$2)}' >> $HLQ/stats/erl
    echo $extern_reuses $total_uses | awk '{print("%f\n", $1/$2)}' >> $HLQ/stats/trl
    echo $intern_reuses $total_uses | awk '{printf("%f\n", $1/$2))' >> $HLQ/stats/irf
    echo $extern_reuses $total_uses | awk '{printf("%f\n", $1/$2)}' >> $HLQ/stats/erf
    echo $intern_reuses $total_uses | awk '{printf("%f\n", $1/$2))' >> $HLQ/stats/trf
    echo $T_LINES_REUSED $T_LINES | awk '{printf("%f\n", $1/$2)}' >> $HLQ/stats/lrl
    echo $T_LINES_REUSED $T_LINES | awk '{printf("%f\n", $1/$2)}' >> $HLQ/stats/lrl
    echo $T_NCSL | awk '{printf("%f\n", $1)}' >> $HLQ/stats/size
    echo $T_NCSL | awk '{printf("%f\n", $1)}' >> $HLQ/stats/size
    echo $infile >> $HLQ/stats/fname
    echo $fname= $infile
    echo $ITL >> $HLQ/stats/itl
    echo $ITL >> $HLQ/stats/etl
    echo $ETL >> $HLQ/stats/abstr
    echo $abstraction= $ABSTRACTION
    if [ -z $MULTICALL ]
then
    echo singlecall >> $HLQ/stats/callgraph
else
    echo multicontrol >> $HLQ/stats/callgraph
fi
}

# Cleanup - delete all temporary files
#-----------------------------------------------
#-----------------------------------------------
cleanup()
#
# Redirect standard out and standard error
exec 2>/dev/null
# Remove temp files
rm $TMPDIR/ccount.out > /dev/null
rm $TMPDIR/cflow.file > /dev/null
rm $TMPDIR/extern_calls > /dev/null
rm $TMPDIR/extern_defs > /dev/null
rm $TMPDIR/extern_counts > /dev/null
rm $TMPDIR/external_ref_counts > /dev/null
rm $TMPDIR/reused_exnodes > /dev/null

104
rm $TMPDIR/intern_calls > /dev/null
dm $TMPDIR/intern_counts > /dev/null
dm $TMPDIR/intern_defn > /dev/null
dm $TMPDIR/intern_ncsl > /dev/null
dm $TMPDIR/intern_refs_counts > /dev/null
dm $TMPDIR/intern_lines > /dev/null
dm $TMPDIR/all_lines > /dev/null
dm exec 2>2
}

# ----------------------------------------
# ----------------------------------------
# Calculate Metrics - driver routine to calculate all metrics
# ----------------------------------------
# ----------------------------------------

Calculate_Metrics()
{
infile=$*;
cfow $infile > $TMPDIR/cflow.file

echo

if [[ $ABSTRACTION = system ]]
then
echo REUSE METRICS FOR SYSTEM
else
echo REUSE METRICS FOR $infile
fi

echo

Reuse_Level_Metrics
Reuse_Frequency_Metrics
total Uses= echo $int_num Uses $extern Uses "+p" | dc

if [[ -z $NO_FDELM ]]
then
    Complexity_Weighting $infile
fi

### uncomment for testing
### Output Stats
### Output_Report
### Cleanup
}

# ----------------------------------------
# ----------------------------------------
# MAIN
# ----------------------------------------
# ----------------------------------------

Parmedit $*
#set -x

echo Internal Threshold Level = $ITL
echo External Threshold Level = $ETL
if [[ -z $ABSTRACTION ]]
  then
  ABSTRACTION=system
fi

if [[ $ABSTRACTION = system ]]
then
  Calculate_Metrics ${CFILE[*]}
elif [[ $ABSTRACTION = file ]]
then
  for inputfile in ${CFILE[*]}
  do
    Calculate_Metrics $inputfile
  done
else
  Function_Reuse_Level_Metrics
fi

exit 0
Appendix C: rl Manual Page
NAME

rl - reuse measurement tool for C.

SYNOPSIS


DESCRIPTION

rl calculates reuse metrics for C source code. It produces a report in the following format:

Reuse Level
---------------------
Internal Functions Reused: <count>
External Functions Reused: <count>
Total Functions Used: <count>

Internal Reuse Level: <fraction>
External Reuse Level: <fraction>
Total Reuse Level: <fraction>

Reuse Frequency
---------------------
Internal References to Reused Functions: <count>
External References to Reused Functions: <count>
Total Number of References: <count>

Internal Reuse Frequency: <fraction>
External Reuse Frequency: <fraction>
Total Reuse Frequency: <fraction>

Lines of Code Reuse Level
---------------------
NCSC (Non-Commentary Source Lines): <count>
Reused NCSC: <count>
Reuse Level for NCSC: <count>

Complexity Weighting
---------------------
Total Non-Commentary Source Lines of code: <count>
Total Non-Commentary Source Lines of reused code: <count>
Complexity Weighting based on size: <fraction>

Reuse Level.
The reuse level metric is a calculation of the amount of C source code that is used more than a given number of times in the given source files.
Internal Functions Reused is a count of the functions defined in the C source files argument and called from at

Page 1
least internal threshold level number of functions (or recursive functions).
External Functions Reused is a count of the functions not defined in the C source files but called at least external threshold level number of times.
Total Functions Used is a count of all functions either defined or called in the C source files. The Internal Reuse Level is the number of internal functions reused divided by the total number of functions used.
External Reuse Level is the number of external functions reused divided by the total number of functions used.
Total Reuse Level is the sum of the internal and external reuse levels. Note that in a system where every function is reused, the total reuse level is 1.

Reuse Frequency.
The reuse frequency metric is based on the number of calls to functions that are called more than a given number of times in the given source files.
Internal References to Reused Functions is a count of calls to functions defined in the C source files and called from at least internal threshold level number of functions (or recursive functions).
External References to Reused Functions is a count of calls to functions not defined in the source files but called at least external threshold level number of times.
Total Number of References is a count of all calls to all functions in the files.
Internal Reuse Frequency is the number of internal references divided by the total number of references.
External Reuse Frequency is the number of external references divided by the total number of references.
Total Reuse Frequency is the sum of the internal and external reuse frequencies.

Lines of Code Reuse Level.
The lines of code reuse level is the number of non-commentary source lines (NCSL) in the reused internal components divided by the total number of NCSL in the system or file.

Complexity Weighting.
The complexity weighting is computed as the sum of the sizes of each function which is reused divided by the sum of the sizes of all C functions. Size is determined by counting Non-Commentary Source Lines (NCSL) of code. The complexity weighting is dependent upon the ccount tool which requires a function delimiter. See the discussion of the -d flag.

OPTIONS
-iltl <internal threshold level>
The internal threshold level is the maximum number of
uses of a component which is defined in the C source files that precede "reuse". The default value is 1.

- etl <external threshold level>
The external threshold level is the maximum number of uses of a component which is defined outside the source listings that precede "reuse". The default value is 1.

- main
Indicates that main() will be included in the count of internal functions. By default, main() is not counted.

- multcall
Specifies that a multcall abstraction will be used when counting reused functions. With this flag, a function can be reused within a single component. This option requires the cscope tool, available only on AT&T Unix V.4.

- high <system|file|function>
Indicates a high level abstraction of system, file or function. The default value is system, in which case the reuse metrics are computed for the system as a whole. If the high level abstraction is file then the metrics are calculated separately for each C source file. If the high level abstraction is function then the metrics are calculated for each function defined in the source file(s). Internal reuse level is currently the only metric available for a high level abstraction of function.

- sysf <filename>
Specifies that all functions listed in the file <filename> should NOT be included in the reuse counts. This feature is intended as a mechanism to prevent inclusion of low-level C system calls such as printf() in the reuse measurements.

- d <delim>
<delim> is the delimiter which precedes each function definition in each C file. This value is necessary to compute the complexity weighting. If this parameter is not given the environment variable FDELMIM will be used. If neither are set then the complexity weighting will be zero. <C source files> The only required argument to rl is one or more names of C source files. The source code should be error free and compilable.

**SUGGESTION**
It is convenient to run rl from your makefile using your source list as input.
SEE ALSO
W. Frakes, "An Empirical Program for Software Reuse"
3rd Workshop on Tools and Methods for Reuse
Syracuse, NY 1990
C. Terry, "Analysis and Implementation of Software Reuse Measurement"
Master's Project and Report, Virginia Tech, NoVa
Falls Church, VA 1993

cflow
ccount
cscope

AUTHORS
Carol Terry cterry@goliat.cs.vt.edu
Bill Frakes frakes@arvis.cs.vt.edu
Chris Fox
Appendix D: C Test Suite
/*
 * t2.c
 * No reused functions, internal or external.
 */

#include <stdio.h>

f1(int n)
{
    static int st=0;
    st += n;
}

main()
{
    int i;
    i = 1;
    f1(i);
}
/**
 t3.c
 1 reused internal func, 2 references to it. No reused external funcs
 */

#include <stdio.h>

f1(int n)
{
 static int st=0;

 st += n;
 if (st < 5)
  f1(n);
}

main()
{
 int i;

 i = 1;
  f1(i);
}
/ * 
  t4.c 
  1 reused internal func and 1 reused external func. 
  2 references to each. 
  */ 

#include <stdio.h> 

f2(int n2) 
{ 
  static int st2=0; 
  st2 += n2; 
  printf("st2 = %d\n",st2); 
} 

f1(int n) 
{ 
  static int st=0; 
  st += n; 
  printf("st = %d\n",st); 
  if (st < 5) 
    f1(n); 
  else 
    f2(n); 
} 

main() 
{ 
  int i; 
  i = 1; 
  f1(i); 
}
/*
  t5.c
  2 reused internal with 4 refs, 3 reused external with 6 refs
*/

#include <stdio.h>
#include <string.h>

f3(int n3)
{
    static int st3=0;
    char txt[10];
    strcpy(txt,"In f3\n");
    printf("The length of txt is %d and contents is:\n %s",strlen(txt),txt);
    st3 += n3;
    printf("st3 = %d\n",st3);
}

f2(int n2)
{
    static int st2=0;
    st2 += n2;
    printf("st2 = %d\n",st2);
    f3(n2);
}

f1(int n)
{
    static int st=0;
    st += n;
    printf("st = %d\n",st);
    if (st < 5)
        f1(n);
    else
        f2(n);
}

main()
{
    int i;
    i = 1;
    f1(i);
    printf("after calling f1 in main\n");
    f3(i);
}
/*
  f2 qualifies for internal reuse only with a multical graph abstract
ion
*/

#include <stdio.h>
#include <string.h>

f3(int n3)
{
  static int st3=0;
  char txt[10];
  
  strcpy(txt,"In f3\n");
  printf("The length of txt is %d and contents is:\n %s",strlen(txt),txt);
  st3 += n3;
  printf("st3 = %d\n",st3);
}

f2(int n2)
{
  static int st2=0;
  st2 += n2;
  printf("st2 = %d\n",st2);
  f3(n2);
}

f1(int n)
{
  static int st=0;
  
  st += n;
  printf("st = %d\n",st);
  f2(n);
  f2(n);
  if (st < 5)
    f1(n);
  else
    f2(n);
}

main()
{
  int i;
  
  i = 1;
  f1(i);
  printf("after calling f1 in main\n");
  f3(i);
}
/*
 * t7.c
 * Has lots of calls to internal function fmany() - test ITL > 1
 * External functions - 5 calls to printf, 2 to strcpy - test ETL > 0
 */

#include <stdio.h>
#include <string.h>

fmany()
{
  printf("inside fmany again\n");
}

f3(int n3)
{
  static int st3=0;
  char txt[10];

  fmany();
  strcpy(txt,"In f3\n");
  printf("The length of txt is %d and contents is:\n  %s",strlen(txt),txt);
  st3 += n3;
  printf("st3 = %d\n",st3);
}

f2(int n2)
{
  static int st2=0;

  fmany();
  st2 += n2;
  printf("st2 = %d\n",st2);
  f3(n2);
}

f1(int n)
{
  static int st=0;

  fmany();
  st += n;
  printf("st = %d\n",st);
  if (st < 5)
    f1(n);
  else
    f2(n);
}

main()
{
  int i;
  char txt[10];

  i = 1;
  fmany();
  f1(i);
  printf("after calling f1 in main\n");
  f3(i);
  strcpy(txt,"etl=1");
}
/*
 * t8.c
 * includes the "/*%" function delimiter for computing the complexity w
 * eighing
 */

#include <stdio.h>
#include <string.h>

/** new function */
fmany()
{
    printf("inside fmany again\n");
}

/** new function */
f3(int n3)
{
    static int st3=0;
    char txt[10];

    fmany();
    strcpy(txt,"In f3\n");
    printf("The length of txt is %d and contents is:\n %s",strlen(txt),txt);
    st3 += n3;
    printf("st3 = %d\n",st3);
}

/** new function */
f2(int n2)
{
    static int st2=0;

    fmany();
    st2 += n2;
    printf("st2 = %d\n",st2);
    f3(n2);
}

/** new function */
f1(int n)
{
    static int st=0;

    fmany();
    st += n;
    printf("st = %d\n",st);
    if (st < 5)
        f1(n);
    else
        f2(n);
}

/** new function */
main()
{
    int i;
    char txt[10];

    i = 1;
    fmany();
    f1(i);
    printf("after calling f1 in main\n");
    f3(i);