Computation of Ripple Effect Measures For Software

Sue Black

A thesis submitted in partial fulfilment of the requirements of South Bank University for the degree of Doctor of Philosophy

September 2001
To Emma, Sam, Oliver, Tubbs and Explory.
Acknowledgements

Firstly, I would like to thank Robin Whitty, my supervisor, for all the time, effort and red ink that he has contributed to the production of this thesis. I am greatly indebted, especially to David Wigg, and also to Lasitha Leelasena for all their work on the REST parser. Thanks to David Mole and all at the Centre for Systems and Software Engineering, past and present, and in the School of Computing, Information Systems and Mathematics who have helped and encouraged me. Thanks also to Mark Harman for rekindling my enthusiasm for research when it has flagged and to British Telecommunications at Martlesham Heath for supporting my research for one year.

Many thanks to all my friends that have supported me during this time, especially: Rosie, Paul, Hazel and Orlando.

Finally, my biggest thanks go to my children Emma, Sam and Olly for all their love and support over the last seventeen years, I couldn’t want for more wonderful children.

Brixton, London
July 2001
Abstract

There are many measures of structural complexity of source code, of which ripple effect is just one. Ripple effect measures the amount which a module or program may affect other modules within a program, or programs within a system, if changes are made. Measurement of ripple effect has been incorporated into several software maintenance models because it shows maintainers the ramifications of any change that they may make before that change is actually implemented. As such, computation of ripple effect provides a potentially valuable source of information. The aim of this thesis is to show that an approximation to Yau and Collofello’s ripple effect algorithm can satisfactorily replace their original algorithm as a measure of structural complexity. The basis of our approach has been to completely reformulate the ripple effect calculation using matrix arithmetic. As well as making the calculation more explicit the reformulation reveals how the algorithm’s structure can be broken down into separate parts. By focusing on the derivation of one particular matrix we find that an approximation may be made, greatly simplifying the calculation. A Ripple Effect and Stability Tool (REST) was created and used to validate our work. Firstly, a comparison of the original and reformulated ripple effect measures from several programs shows them to be highly correlated. Secondly, a case study is used to explore the link between ripple effect and maintainer’s intuition of the impact of code changes. Perhaps unsurprisingly, this link appears to be less than clear-cut.
# Table of Contents

## Introduction

1 Software Measurement

1.1 Software Measurement Basics ........................................ 5

1.1.1 Internal and External Attributes ................................. 5

1.1.2 Direct and Indirect Measures ..................................... 6

1.1.3 Scale Types .......................................................... 7

1.2 Typical Software Product Measures .................................. 8

1.2.1 Measuring Length .................................................. 8

1.2.2 Measuring Functionality .......................................... 9

1.2.3 Measuring Complexity ............................................ 9

1.3 Complexity Measures ................................................ 10

1.3.1 Measuring Data Flow Structure ................................ 11

1.3.2 Measuring Control Flow Structure ............................... 15

1.4 McCabe's Cyclomatic Complexity .................................. 16

1.5 Summary ............................................................... 20

2 Ripple Effect Analysis

2.1 Introduction ............................................................ 22

2.2 Yau and Collofello .................................................... 23

2.2.1 Code Level Logical Stability .................................... 24

2.2.2 Design Level Logical Stability ................................... 26

2.3 Ripple Effect and Software Maintenance ............................ 27

2.4 Automating Ripple Effect ............................................. 29

2.4.1 Tools Which Produce a Ripple Effect Measure ............... 29

2.4.2 Tools Which Trace Ripple Effect Through a System ....... 32

2.5 Other Work ............................................................ 33

2.6 Summary ............................................................... 35
3 Ripple Effect as a Matrix Product

3.1 Intramodule Change Propagation

3.2 Intermodule Change Propagation

3.3 Complexity and Logical Stability

3.4 Computing Ripple Effect for an Example Program

3.5 Summary

4 Deriving an Approximation

4.1 Decomposition of Matrix $Z_m$

4.2 Approximated Matrix $D$

4.3 Matrix $C$

4.4 Versions of the McCabe Complexity Factor

4.5 Summary

5 Ripple Effect and Stability Tool

5.1 The Need for Fully Automated Computation of Ripple Effect

5.2 The REST Software Tool

5.2.1 Parser

5.2.2 Listfuns

5.2.3 Funmat

5.2.4 Ripple

5.3 Summary

6 Validation of the Approximation Algorithm

6.1 Programs with Identical Ripple Effect

6.1.1 Reasons for Identical Ripple Effect

6.1.2 Funmat5.c

6.1.3 Conv.c

6.1.4 Quadratic.c

6.1.5 Genscrip.c

6.1.6 Tamlize.c

6.2 Programs with Differing Ripple Effect

6.2.1 Reasons for Differing Ripple Effect

6.2.2 Gensym.c

6.2.3 Error.c

6.2.4 Bits.c

6.2.5 Arippletest.c

6.2.6 Fcomp.c

6.2.7 Metrics.c

6.3 Spurious 1s

6.4 McCabe Variants and their correlation

vi
Introduction

Ripple effect has an intuitive appeal. Imagine a stone being thrown into a pond: it makes a sound as it enters the water and causes ripples to move outwards to the edge of the pond. It is reasonably easy to transfer this image to source code. The stone entering the water is now a hypothetical change to the source code of a program, the effect of the change ripples across the source code via data flow. As part of software development or maintenance we may want to ask questions about the program such as:

"How much ripple is there?"

"Which parts of the program affect other parts the most?"

The research described in this thesis provides a specific practical solution to these questions. Typically, 70% [Ben90] of software development budgets are spent on maintenance, thus its importance in the field of software engineering cannot be denied. Any measures or tools which can assist maintainers in their role by speeding up the rate at which changes can be made, or enabling maintainers to make better informed decisions on code changes can thus make an important contribution. As all types of maintenance involve making changes to source code, ripple effect can be used to help maintainers by highlighting modules which may cause problems during the maintenance process. Ripple effect can show the maintainer what the effect of that change will be on the rest of the program or system. It can highlight modules with high ripple effect as possible problem modules, show the impact in terms of increased ripple effect or look at the ripple effect of a program and its modules before
and after a change to ascertain whether the change has increased, or perhaps decreased, the stability of the program. Maintenance is difficult [vZE93] because it is not clear where modifications have to be made or what the impact will be on the rest of the source code once those changes are made. The ripple effect can be used to help maintainers with the latter. Ripple effect, along with many other metrics, is not the answer to all maintainers' problems, but used as part of a suite of metrics it can give maintainers useful information to make their task easier.

An algorithm for computing ripple effect was developed by Yau and Collofello in 1978 [YCM78] and developed over several years. Several attempts have been made at using the algorithm to construct a tool to produce fast and accurate ripple effect measurements, none of which have completely succeeded. Our contribution has been to reformulate the ripple effect algorithm using matrix arithmetic and to use this reformulated algorithm to produce a software tool: REST. The reformulation makes the calculation more explicit and reveals how the algorithm's structure can be broken down into separate parts. By focusing on the derivation of one particular matrix: $D$, an approximation has been made simplifying the calculation.

The structure of this dissertation is as follows:

Chapter 1 describes the field of software measurement focusing on software complexity measurement. The final section is a description and criticism of McCabe's cyclomatic complexity measure which is used as part of the ripple effect measure. The validity of McCabe's measure as a measure of complexity is discussed to provide a background to some variations which we propose in 4.4. These variations aim to counteract the effect of our approximation.

Chapter 2 looks at the evolution of ripple effect and logical stability measures. Yau and Collofello have made the major contribution in this area with their code level and design
level measures. Ripple effect is included in several software maintenance models as part of the maintenance lifecycle. We take a look at these models and show where ripple effect measurement fits in. Finally, other research in the area of ripple effect measurement is described.

Chapter 3 gives a precise definition of the computation of ripple effect. Two fundamental ideas in its computation are intramodule and intermodule change propagation. This chapter gives a detailed description of what they are and how they may be computed using two matrices, which we call $Z$ and $X$ respectively. We calculate ripple effect for an example program to clarify the use of the reformulated algorithm in practice.

Chapter 4 concentrates on matrix $Z$. We find that we can write $Z = \sum (A + D)^i$ where $i =$ number of variables within a module, for certain matrices $A$ and $D$. $A$ is easy to compute; $D$ is at the heart of the difficulty in computing ripple effect. We therefore propose to replace $D$ with a simplification $D'$. Although $D'$ can potentially exaggerate intramodule change propagation, in practice we discover this effect to be anodyne. The approximated version of $D$, $D'$, is described with examples which illustrate the effect that control flow has on its accuracy. Matrix $C$, the complexity factor, is shown in several variations which are investigated as possible counterbalances to the inaccuracy of matrix $D'$.

Chapter 5 is a description of REST (the Ripple Effect and Stability Tool) which has been built at the Centre for Systems and Software Engineering. The need for the tool, its place in relation to other ripple effect measurement tools, and its contribution towards our research are explained. The individual components making up REST are detailed with a description of how they all link together to produce ripple effect measures.

Chapter 6 describes in detail the eleven programs which have been used to validate the reformulated algorithm. These programs are divided into two groups: those programs
where the Yau and Collofello ripple effect is identical to the approximated ripple effect, and those where it is not. Each group of programs is followed by a table showing the Pearson correlation coefficient and a discussion of the results for that group. There is also some discussion of the number of spurious 1s within $D'$ for each program and whether this has an effect on the accuracy of the approximated ripple effect measure.

Chapter 7 looks at the product-process link. A case study is described of a mutation testing software tool which is used to show the correlation between the developer's intuition of the ripple effect and the actual ripple effect measure. Results show that the main reason for disparity between approximated and predicted ripple effect is that small modules with few variable definitions and high amounts of intermodule change propagation have a higher approximated ripple effect than large modules with a similar amount of intermodule change propagation. Predicted ripple effect for the smaller complex modules is lower than for the larger complex modules possibly due to the reasoning that a large module will affect other modules more than a small module.

Chapter 8 is the concluding chapter in which we summarise and conclude our dissertation and make some suggestions for further work.
Chapter 1
Software Measurement

Software measurement as a software engineering discipline has been around now for some thirty years [Zus98]. Its purpose is to provide data which can be used either for assessment or prediction during the software lifecycle. Typically it is used for assessment either during the initial development of software, or during maintenance of software at a later date. Software measurement is now a very complex subject. This chapter aims to provide a general software measurement background for our subsequent discussion of ripple effect and logical stability measures. We begin with a short general introduction to software measurement followed by the most straightforward example: measuring software size. Software complexity is then introduced and explained in detail. Finally McCabe's cyclomatic complexity measure is described as it is used by Yau and Collofello as part of their ripple effect algorithm. The validity of McCabe's measure as a measure of complexity is discussed to provide a background to its variations which we propose in 4.4. These variations aim to counteract the effect of not including intramodule control flow in our approximation.

1.1 Software Measurement Basics

There are many aspects of software that we may wish to measure: source code length, time taken to write source code, or the price at which the executable source code is eventually
sold. In each case we are measuring a specific attribute of a particular entity: length and price are attributes of the entity source code, while time is an attribute of the process which produced it. Thus length and price are product attributes, while development time is a process attribute. But, this distinction is not always clear cut: someone buying software as a resource will regard its price as a resource attribute. Any entities that we may measure can be subdivided into [BF90]:

- **Processes** - any software related activities e.g writing source code.
- **Products** - any artefacts which arise from processes e.g. source code.
- **Resources** - any resource which is input to a process e.g. personnel.

### 1.1.1 Internal and External Attributes

The attributes of entities can be broken down into two categories: internal and external [BF90]. Internal attributes are those which can be measured purely in terms of the entity itself; external attributes can only be measured with respect to how the entity relates to its environment. For example, size of code is an internal attribute whilst reliability of code is an external attribute. It is not always easy to distinguish between an attribute being internal or external. Within ISO 9126 [iso91] it is proposed that the quality of a software product may be evaluated by the following attributes: functionality, reliability, usability, efficiency, maintainability and portability. It is not immediately apparent that functionality is the only internal attribute, although this is in fact the case as all the others are dependent on the environment.

### 1.1.2 Direct and Indirect Measures

Measures can be classified according to whether the measure is a direct or indirect measure of an attribute of some entity. Direct measures of an attribute are those for which
the measurement does not depend upon any other attribute. Examples of direct software measures are: length of source code, duration of testing process. Indirect measurement of an attribute is measurement which also involves the measurement of other attributes. Examples of indirect measurement are [FP96]:

\[
\text{programmer productivity} = \frac{\text{LOC produced}}{\text{Person months of effort}}
\]

\[
\text{Module defect density} = \frac{\text{Number of defects}}{\text{Module size}}
\]

1.1.3 Scale Types

A factor which needs to be taken into consideration when using software measures is their scale types. Each measure has a scale type determined by the type of relationship the measure is capturing. There are five commonly used scale types [FP96]:

- **Nominal measures** do not capture any concept of order with respect to the attribute, they simply classify entities. An example of a nominal measure is colour.

- **Ordinal measures** classify entities with respect to the attribute and also place them in order. An example of a ordinal measure is the scores given in a contest.

- **Interval measures** capture the order of entities with respect to the attribute and some idea of difference between the entities with respect to the attribute. They have no absolute zero, allowing the possibility of positive and negative values. An example of an interval measure is temperature in degrees Fahrenheit.

- **Ratio measures** have the properties of interval measures and also have an absolute zero. This means that it is possible to use ratio when referring to these measures e.g. it is reasonable to say "...it took three times as long to to write program 1 as program 2", because elapsed time is measured on a ratio scale.
• Absolute measures count the number of occurrences of the attribute in question. An example of an absolute measure is the number of lines of code in a program.

These scale types have been presented in ascending order with the weakest being nominal and the strongest being absolute. It is often a goal in measurement to obtain as strong a type as possible; the development of a stronger scale generally takes place with a better understanding of the attribute being measured.

1.2 Typical Software Product Measures

When we consider measuring software we can look at it from several viewpoints [FP96]. Probably the most intuitive measure is length e.g. lines of code (LOC) which measures the physical size of the code. We can also measure functionality in terms of what the user actually gets from the code, and complexity e.g. the amount of resources required for the solution of a problem. The size of at least three different outputs from the software lifecycle can be measured: specification, design and source code. Measurement of specification size can give us an indication of the likely size of the design and source code. Measurement of the design can give an indication of the likely size of the source code. Thus measurement of length can be useful in a predictive manner throughout the software lifecycle. We may also wish to assess these products e.g. for conformance (to a standard) and measurement will again be important in this.

1.2.1 Measuring Length

Number of lines of code is the most commonly used measure of source code program length. But, even with such a simple measure there is room for ambiguity. It must always be made apparent which lines are included, i.e. whether to count comment lines, blank lines and data declarations along with program statements. It also needs to be made clear whether a line
containing several separate instructions is counted as one line or one line per instruction. The most widely accepted definition used for measuring LOC is NCLOC (Non-Commented Lines Of Code) which was defined by Hewlett Packard [GC87]. NCLOC includes all lines within a program except blank lines and comment lines. Length of designs and specifications can also be measured by counting atomic objects such as processes and data stores in a data-flow diagram or type declarations and predicates in a Z specification.

1.2.2 Measuring Functionality

Functionality is often measured to provide effort and duration estimates from the specification or design of a system. Albrecht's function points [Alb79] are probably the best known measure of functionality used. Computation of function points for a particular system involves counting atomic objects such as inputs, outputs and files, and then using a technical complexity factor to produce a final function point count. There are several problems with function point analysis as detailed in [FP96]. Other functionality measures include some cost drivers within COCOMO [Boe81] and DeMarco's bang metric [DeM82].

1.2.3 Measuring Complexity

When discussing complexity as a measure of program size we first need to decide which of the following types we are interested in [FP96]:

- Computational complexity the complexity of the underlying problem.

- Algorithmic complexity the complexity of the algorithm which has been implemented.

- Structural complexity the complexity of the software used to implement the algorithm.
• Cognitive complexity the effort required to understand the software.

Ripple effect and logical stability are measures of structural complexity. The next section looks at complexity measures in more detail, focusing on structural complexity.

1.3 Complexity Measures

Early measurement of software complexity focused entirely on source code with the simplest complexity measure being LOC. In 1983 Basili and Hutchens [BH83] suggested that LOC be used as a baseline or benchmark to which all other complexity metrics be compared i.e. an effective metric should perform better than LOC so LOC should be used as a ‘null hypothesis’ for empirical evaluation. Much empirical work has shown it to correlate with other metrics [She93], most notably McCabe’s cyclomatic complexity which is discussed in more detail in section 1.4. The earliest code metric based on a coherent model of software complexity was Halstead’s software science [Hal71]. Early empirical evaluations produced high correlations between predicted and actual results but later work showed a lower correlation. Bowen [Bow78] found only modest correlation, with software science being outperformed by LOC. According to Shepperd [She93] the most important legacy of software science is that it attempts to provide a coherent and explicit model of program complexity as a framework within which to measure and make interpretations. Software science is also important because, since it deals with tokens, it is fairly language independent.

After a concentration on code level measurement for some years, focus widened to include measurement during the earlier stages of the software development lifecycle. Design level metrics can in theory be obtained much earlier in the development of a project thus providing information which can be used for more informed resource management.

Structural complexity of software can be broken down into the following types:
- **Data flow structure** the way that data flows through a program and its behaviour as it interacts with the program.

- **Control flow structure** the order in which program instructions are executed taking into account whether there are any loops or branches.

- **Data structure** the organisation of the data itself, independent of the program. Data structure will not be discussed further in this thesis (see [FP96] page 317).

The ripple effect measure is fundamentally concerned with the data flow structure of a program. Control flow issues are also relevant to its computation and this will be found to be a key issue in our approximation of ripple effect measurement.

### 1.3.1 Measuring Data Flow Structure

As mentioned previously, data flow structure concerns the way in which data flows around a program. An important factor to consider when looking at the data flow structure is the scope of variables. In some programming languages, a variable within a program can be local to a procedure/function or global i.e. be available for use throughout the program.
Some programming languages also allow for recursive definitions of local and global, for example Pascal which allows nesting of procedures [BC00]. When considering data flow within a program, scope of variables is important as it can dramatically affect the effect that a variable on other variables across a program.

It can be seen from Figure 1.1a that variable \( x \) is declared locally within module 3 and can therefore only be used within module 3. Figure 1.1b shows variable \( x \) declared within module 2. As module 3 is nested within module 2, the variable \( x \) is also available for use by module 3. Figure 1.1c shows variable \( x \) declared at the top of the program outside of modules 1, 2, and 3. Variable \( x \) is therefore available for use by all modules and is hence considered global.

As well as considering the scope of a variable we also need to look at data flow from two different perspectives, namely: intramodule and intermodule data flow. Intramodule data flow concerns data flow within a module. A module is taken here to mean a function or procedure but can be used at different levels of abstraction e.g. a program within a systems or a subsystem within a larger system. A module is defined by Yourdon [You79] as:

"a contiguous sequence of program statements bounded by boundary elements, having an aggregate identifier".

In Figure 1.2 there is intramodule data flow between \( x \) and \( y \) as \( y \) is being assigned the value of \( x \).
Intermodule data flow concerns the flow of data between modules. Figure 1.3 shows intermodule data flow between the main program where $z$ is declared as a global variable and module 1 where the value of $z$ is being assigned to $y$.

There are several well known measures of data flow structure. These include coupling, cohesion and information flow. Coupling and cohesion were first proposed in 1974 [SMC74] as a measure of design quality. Coupling concerns the degree of interdependence between modules [You79], it is dependent on the type of connections and how complicated they are. Thus coupling is a measure of intermodule data flow. Cohesion is a measure of the relationship of the elements within a module: ideally a module should perform a single function [TZ81]. Yourdon and Constantine proposed seven classes of cohesion which range from:

- Functional - the module performs a single well defined function.

through to

- Coincidental - the module performs more than one function and they are unrelated.

As modules may exhibit more than one type of cohesion they are categorised by the lowest type that they exhibit. When measuring the coupling and cohesion of a system it is common to aim for high cohesion and low coupling.
Information flow [HK81] is a measure of the total level of information flow between individual modules and the rest of a system. The two fundamental concepts within the information flow measure are \textit{fan-in} and \textit{fan-out}.

\textbf{Definition:} \textit{The fan-in (fan-out) of a procedure is the number of local flows terminating at (emanating from) that procedure.}

Information flow complexity was defined by Henry and Kafura [HK81] as:

\[
\text{Information flow complexity } (M) = \text{length}(M) \times (\text{fan-in}(M) \times \text{fan-out}(M))^2
\]

This has been refined by Shepperd [She90] to produce the following measure (the name is from [FP96, p.203]):

\[
\text{Shepperd complexity } (M) = (\text{fan-in}(M) \times \text{fan-out}(M))^2
\]

which Shepperd claims is an improvement over the original Henry and Kafura measure because it eliminates the control flow element present in the original. Shepperd disregards module length and indirect information flow in his measure along with the distinction between local and global information flow. This is to eliminate the blurring of information and control flow. Shepperd complexity is found to correlate highly with development time, whereas Henry and Kafura's information flow has no such relationship.

Coupling and information flow relate to intermodule data flow, cohesion relates to intramodule data flow. The ripple effect measure concerns itself with both intramodule and intermodule data flow. This will be described and explained in greater detail in chapter 2. The measurement of ripple effect also traditionally involves a control flow complexity component namely: McCabe's cyclomatic complexity. Control flow measures are described in the next section followed by a discussion of McCabe's measure and its validity as a measure of control flow complexity.
read(x); L := Ø;
repeat
    if odd(x) then
        x := 3x + 1;
    add x div 2 to list L
until L contains a repetition;
if x ≠ 1 then
    write("x is a counter-example")
Figure 1.4: Flowgraph from [Whi97]

1.3.2 Measuring Control Flow Structure

Control flow structure is concerned with the order in which program instructions are executed, taking into account whether there are any loops or branches. The control flow of a module can be modelled using directed graphs called flowgraphs. Figure 1.4 shows a flowgraph for a module which demonstrates the three fundamental programming concepts: sequence, selection (branching) and iteration (looping). There are many measures which are associated with control flow structure, the best known of which is McCabe's cyclomatic complexity.

We are interested in measuring control flow complexity because ripple effect has a control flow complexity component. The original Yau and Collofello algorithm uses McCabe's cyclomatic complexity to add a complexity weighting to their ripple effect measure. In our reformulated algorithm control flow inside modules (intramodule change propagation) is approximated and is liable to be exaggerated, we have therefore suggested some variations.
of the complexity component to counteract any effect this may have had. The variations are discussed in more detail in 4.4

1.4 McCabe's Cyclomatic Complexity

In McCabe's 1976 paper [McC76] he describes his cyclomatic complexity measure as:

"...a graph theoretic complexity measure".

He was trying to develop a mathematical technique based on program control flow which would help software engineers produce modularised code which was both testable and maintainable. McCabe could see no obvious relationship between length of module and module complexity so suggested that the number of control paths through a module might be a better indicator.

For a flowgraph F cyclomatic complexity can be defined as [McC76]:

\[ v(F) = e - n + 2 \]

where F has e arcs and n nodes. For example, for the flowgraph in Figure 1.4 we have:

\[ v(F) = 10 - 8 + 2 \]

\[ = 4 \]

Cyclomatic complexity can also be informally defined as [McC76, p.315]: "number of predicates plus one"; the program in Figure 1.4 has two selection decisions and one iteration decision. McCabe's measure has been widely criticised [Lea90], [Pra84], [Mye83], [Han78], [Eva83] since its appearance in 1976. This may be partly due to the fact that it is not appropriate for all third generation languages: McCabe was originally thinking in terms of a complexity measure for Fortran source code. With his cyclomatic complexity measure he had two objectives:
• To predict the effort of testing data and therefore identify appropriate decompositions of software in modules

• To predict the complexity related characteristics of the resultant software

Shepperd provides an excellent critique of cyclomatic complexity in [She88]. He criticises cyclomatic complexity for being based on poor theoretical foundations and being outperformed as a measure of general complexity by lines of code in one third of cases. He suggests that the number of control flow paths through a module would be a better indicator of complexity. Fenton and Pfeeger [FP96] point out that cyclomatic complexity is objective and useful when counting linearly independent paths, but do not think that it gives an accurate picture of measure program complexity. They recommend that it be used as an indicator of testability and maintainability, and perhaps used as part of a quality assurance exercise. McCabe suggests that a cyclomatic complexity of greater than 10 in any one module means problems, whereas Grady [Gra94] who discovered a relationship between the number of updates to software required and cyclomatic complexity thought that this limit should be set at fifteen. During the quality assurance project on the software used for the channel tunnel, modules were rejected if their cyclomatic complexity was greater than twenty, or the number of statements was greater than fifty [Ben90].

Myers [Mye83] posits that intuitive complexity does not align with cyclomatic complexity: it is possible to calculate a higher complexity measure for a program of lesser complexity, thus demonstrating that cyclomatic complexity fails even to capture complexity on the ordinal scale. He gives an example of three statements:

A: If \((x=0)\) THEN...

ELSE...
B: If (x=0) AND (y=0) THEN...
    ELSE...

C: IF (x=0) THEN
    IF (y=1) THEN...
    ELSE...
    ELSE...

Myers proposes that the intuitive control flow complexity increases from A to B to C, i.e. C is the most complex and A the least complex, whereas the cyclomatic complexity measure for A and B is in fact the same. He suggests that an improved cyclomatic complexity measure would also include a count of conditions within predicates. In fact McCabe had already addressed this problem. McCabe states in [McC76, p. 314] that 'In practice, compound predicates such as IF, "C1 AND C2" THEN are treated as contributing 2 to complexity, since without the connective AND we would have IF C1 THEN IF C2 THEN which has two predicates. For this reason and for testing purposes, it has been found to be more convenient to count conditions instead of predicates when calculating complexity.' Hansen [Han78] suggests that an operator count should also be included for this to be a true complexity measure. Evangelist [Eva83] gives an example where there is high cyclomatic complexity but low intuitive complexity:

```plaintext
if(a1==0) b1=0; else b1=1;
if(a2==0) b2=0; else b2=1;
   .
   .
   .
if(an==0) bn=0; else bn=1;
```
It can be seen that this program would have a high cyclomatic complexity but a low computational and intuitive complexity. Evangelist goes on to state [Eva83, p.58] that: "...the obvious solution is to represent software complexity as a combination of quantities derived from both static and dynamic program properties".

Basili and Reiter [BRWR79] sought to find out whether automatable software measurement was worthwhile. Measurement experiments were carried out on nineteen different implementations of the same software development project using several variations of McCabe's cyclomatic complexity. They came to the conclusion that CASE statements should be weighted differently from other control flow features as they contribute less to the overall complexity of the module. Prather [Pra84] considers cyclomatic complexity too elementary to reflect the intricacies of program complexity. His fundamental objections to cyclomatic complexity are that it is relatively insensitive to program restructuring, it correlates too closely with LOC and it takes no account of nesting levels. He gives examples which illustrate his objections and suggests a set of axioms that a general complexity metric must satisfy:

**Axiom 1** The complexity of the whole must not be less than the sum of the complexities of the parts

**Axiom 2** The complexity of a selection must be greater than the summed complexity of all the branches

**Axiom 3** The complexity of an iteration must be greater than the summed complexity of the iterated parts

This paper spawned a whole theory of axiomatic control flow complexity measurement, described in [FP96].
Leach [Lea90] describes the results of an analysis of a software system that underwent several revisions. Measurements of Halstead's Software Science Effort, McCabe's cyclomatic complexity and coupling analysis were performed after each revision of the software. He found that cyclomatic complexity of modules did not change much over several revisions. One of his conclusions is that the dictum that a cyclomatic complexity for each module of not greater than ten does not apply when looking at 'switch' statements in C.

There are many negative criticisms of McCabe's measure, leaving one with the impression that cyclomatic complexity is not a good measure of general complexity. It must be taken into consideration though that it was one of the first software measures put forward and as such, we cannot expect it to have taken into account all of the advances that have taken place in measurement in the last twenty or so years. It does give us some information about the complexity of software, and can then be used as long its limitations are appreciated. In any case, it is an integral part of the original ripple effect algorithm; we have preserved it in the reformulation which we present in the next chapter and the above discussion provides some necessary context for this decision. Moreover, in Chapter 4 we propose some variants to McCabe's original measure. These should be seen in the context of variations such as those proposed by Myers and by the axiomatic complexity theorists, although our aim is primarily to counteract the effects of our approximation of intramodule flow.

1.5 Summary

In this chapter some of the fundamental concepts of software measurement have been introduced; measurement of software size and complexity have been explained with particular emphasis on software complexity. The final section of this chapter has been devoted to a description and criticism of McCabe's cyclomatic complexity, a complexity measure which is used as part of the ripple effect measure. Cyclomatic complexity is used as part of the
original formulation of the ripple effect measure as a 'measure of the individual complexity of a module' [YC80]. We discuss it in this chapter to provide a background to its variations which are proposed in 4.4.
Chapter 2

Ripple Effect Analysis

This chapter looks at the evolution of ripple effect and logical stability measures. Yau and Collofello have made the major contribution in this field, but prior to their work Haney and Myers used the term 'ripple effect'; their work is described in section one. Section two describes Yau and Collofello's contribution to ripple effect analysis. Their algorithm evolves through several stages facilitating measurement of ripple effect at code and design level. Section three shows where computation of ripple effect fits in to the software maintenance lifecycle. Section four looks at automation of ripple effect measurement and section five details other relevant research carried out in this field.

2.1 Introduction

The ripple effect measures impact, or how likely it is that a change to a particular module is going to cause problems in the rest of a program. It can also be used as an indicator of the complexity of a particular module or program. Ripple effect was one of the earliest metrics concerned with the structure of a system and how its modules interact [She93]. The first mention of the term ripple effect in software engineering is by Haney in 1972 [Han72]. He uses a technique called 'module connection analysis' to estimate the total number of changes needed to stabilise a system. Myers [Mye80] uses the joint probability of connection between
all elements within a system to produce a program stability measure. A matrix is set up to store the weighting of each possible connection within a system, then another matrix is derived estimating the joint probability density for any two states in the first matrix. The limit probability vector is found using these matrices and used to calculate the stability of the system. Soong [Soo77] used the joint probability of connection of all elements within a system to produce a program stability measure. Haney, Myers and Soong's methods are all measures of probability, the probability of a change to a variable or module affecting another variable or module. Yau and Collofello's ripple effect uses ideas from this research but their ripple effect is not a measure of probability.

2.2 Yau and Collofello

When Yau and Collofello first proposed their ripple effect analysis technique in 1978 [YCM78] they saw it as a complexity measure which could be used during software maintenance to evaluate and compare various modifications to source code. Ripple effect was defined by Haney [Han72] as:

"The phenomenon by which changes to one program area have tendencies to be felt in other program areas"

It is split by Yau and Collofello into two aspects:

Logical: identification of program areas requiring additional maintenance to ensure consistency with an original change.

Performance: analysis of changes to one program area which may affect the performance of other program areas.

The technique did not provide proposals for modifying a system, but rather was applied after a number of maintenance proposals had been generated. The complexity could then
be computed for each modification and the best proposal selected from both a logical and a performance perspective.

Computation of logical ripple effect involved using error flow analysis. All program variable definitions involved in an initial modification represented primary error sources from which inconsistency could propagate to other program areas. Identification of affected program areas could then be made by internally tracking each primary error source and its respective secondary error sources within the module to a point of exit. At each point of exit a determination would be made as to which error sources propagated across module boundaries. Those that did became primary error sources within the relevant modules. Propagation continued until no new secondary error sources were created. The analysis is split into two stages: lexical analysis and computation of ripple effect.

2.2.1 Code Level Logical Stability

This work was carried further in 1980 to produce a logical stability measure (the algorithm given in [YC80] is reproduced in Appendix B). Logical stability is defined [YC80, p. 547] as:

"a measure of the resistance to the expected impact of a modification to the module on other modules within the program".

In [YC80] a software maintenance process is identified of which accounting for ripple effect is Phase 3 (Figure 2.1). Logical ripple effect is now split into two more easily comprehensible aspects: intramodule change propagation and intermodule change propagation. Intramodule change propagation was used to identify the set $Z_{ki}$ of interface variables which are affected by logical ripple effect as a consequence of modification to variable definition $i$ in module $k$. This requires an identification of which variables constitute the module's interfaces with other modules and a characterisation of the potential intramodule change
Figure 2.1: A methodology for software maintenance

propagation among the variables inside the module. Interface variables are defined as: global variables, output parameters and variables utilised as input parameters to called modules. Intermodule change propagation is then used to compute the set $X_{kj}$ consisting of modules involved in intermodule change propagation as a consequence of being affected by interface variable $j$ of module $k$.

$X_{kj}$ is calculated by first identifying all of the modules for which $j$ is an input parameter or global variable. For each of these modules the intramodule change propagation emanating from $j$ is then traced to the interface variables within the module. The modules involved in the intermodule change propagation as a consequence of modifying variable $i$ of module $k$ can then be represented by the set $W_{ki}$. The logical complexity $LCM_{ki}$ of each variable $i$ in every module $k$ is then computed using McCabe's Cyclomatic complexity. The probability that a particular variable definition $i$ of a module $k$ will be selected for modification is then estimated as:
26

(number of variable definitions in the module)

The product of the probability with the \( LCM \) for each variable definition \( i \) gives the logical ripple effect for the module \( k \), \( LRE_k \). The logical stability measure for module \( k \) denoted \( LS_k \) is the reciprocal of \( LRE_k \).

2.2.2 Design Level Logical Stability

In the eighties the general emphasis for software measurement extended from source code measurement to measurement of design. The thinking behind this was that as design measurement gives feedback earlier in the software lifecycle, problems could be identified and eliminated or controlled before the source code was actually written, thus saving time and money.

Yau and Collofello published a paper applying the same ideas that they had used in producing their code level stability measure [YC80] to produce a design level stability measure [YC85]. The design measure analyses the module invocation hierarchy and use of global data referenced or defined in modules to produce the design stability of a program. The main difference between code level stability and design level stability is that the design stability algorithm does not consider intramodule change propagation. It produces a measure of ripple effect between modules without taking into account what happens inside them. This presupposes that information about parameters passed between modules, global variables etc. is already known. Yau and Collofello recommended that their measure be used to compare alternative programs at the design phase and to identify which portions of the program may cause problems with ripple effect during the maintenance phase. It will be seen that our approximated computation of code-level ripple effect is based on making a general assumption about intermodule flow. It might therefore be seen as sitting mid-way between Yau and Collofello's original algorithm and their proposal for a design level measure.
2.3 Ripple Effect and Software Maintenance

Software maintenance has been classified into four types [Pre94]:

- Perfective maintenance - to alter functionality
- Adaptive maintenance - to adapt software to changes in its environment
- Corrective maintenance - to correct errors
- Preventative maintenance - to update software in anticipation of future problems

Ripple effect can highlight modules with high ripple effect as possible problem modules which may be especially useful in preventative maintenance. It can show the impact in terms of increased ripple effect during perfective and adaptive maintenance where the functionality of a program is being modified or its environment has changed. During corrective maintenance it may be helpful to look at the ripple effect of the changed program and its modules before and after a change to ascertain whether the change has increased, or perhaps decreased, the stability of the program.

It is generally believed that there is a strong link between software maintenance and ripple effect. Computation of ripple effect and logical stability of a module are based on a subset of the maintenance activity: a change to a single variable definition within a module [YC80]. Regardless of the complexity of the maintenance activity being performed, it basically consists of modifications to variables within modules of code. Logical stability is computed based on the impact of these modifications. It can be used to predict the impact of primitive modifications on a program and thus be used to compute the logical stability of modules with respect to the primitive modifications. The effect of modification may not be local to the module but may affect other parts of the program, there is a ripple effect from the location of the modification to other parts of the program that are affected by the
modification. If the stability of a program is poor the impact of any modification is large, hence the maintenance cost will be high and reliability may also suffer.

Several software maintenance models have been proposed in the past, Boehm’s model[Boe87] consists of three major phases:

- understanding the software
- modifying the software
- revalidating the software

These are the fundamental activities of the software maintenance process. With Yau’s model shown in Figure 2.1 [YC80], there is the introduction of impact analysis into the lifecycle.

The model consists of four phases, and includes analysis and monitoring of the impact of change at phase three “accounting for ripple effect”. The aims of the model were to assist in achieving cost effective software maintenance and the development of easily maintainable software. Ripple effect is measured by Yau et al [YCM78] as part of a maintenance technique with which maintenance practitioners can understand the scope of changes made to their programs. Yau et al found that applying a maintenance technique based on ripple effect analysis gave benefits including: smoother implementation of changes, the injection of fewer faults, less structural degradation, a decrease in the growth of complexity and an extension to the operating life of the software.

The Pfleeger and Bohner model (Figure 2.2) [PB90] has six phases, the main difference from Yau’s model being that it includes “analyse software change impact” at phase two i.e. much earlier in the lifecycle. The feedback paths in the SADT model indicate attributes that must be measured, the results are then assessed by management before the next activity is undertaken. The metrics act as a controlling mechanism in the progression from existing
system and change requests, to new system. More detailed information on our approximated ripple effect algorithm and software maintenance can be found in [Bla01].

2.4 Automating Ripple Effect

Automation of ripple effect can focus in two directions: the computation of ripple effect measures; the tracing of ripple effect of variables through a program or a system. Tools produced for both of these categories of ripple effect are described in the next two sections. Hsieh's tool described in Section 2.4.1 both produces a measure and traces ripple through programs.

2.4.1 Tools Which Produce a Ripple Effect Measure

A prototype tool for ripple effect analysis of Pascal programs is introduced by Hsieh in [Hsi82]. The pseudocode algorithms used to produce the tool are presented and explained in detail. The tool consists of three subsystems: an intramodule error flow analyser, an
intermodule error flow analyser and a logical ripple effect identification subsystem. They found that they could not identify primary error sources automatically, thus some user input was required. The intramodule error flow analyser collects information about variable modification and use and the relationship between those modifications and uses. The intermodule error flow analyser computes summary data flow information for each module. Then, the logical ripple effect identification subsystem traces the impact of each variable identified by the user. The tracing phase consumes a large amount of computation time, which according to [YC84] made it infeasible in some cases. Hsieh states that: “The experience we gathered using this prototype system indicates that the logical ripple effect analysis technique can be a powerful tool...” [Hsi82, p. 151]. Unfortunately no details are given of any measures produced by the tool in [Hsi82], but they are referred to for comparison by Yau and Chang in [YC84].

Yau and Chang [YC84] found that techniques for performing ripple effect analysis were taking too much computation time to be practicable for large programs. They presented a new algorithm [YC84] which was put forward as being much faster at computing logical stability than previous versions. Processor time for computation of logical stability for six programs was compared with the processor time using Hsieh's tool (see above). Their algorithm does not include information from the intramodule phase as they felt that disregarding this information simplified the problem and also simulated the environment of the program design phase. Logical stability and speed of calculation for Pascal programs between 684 and 1744 lines long were compared (shown in Table 2.1). Five of the six programs are selected from programming assignments submitted by Computer Science students, the other program (number 2) is a sample program from an Air Force agency which has been translated by hand from Jovial to Pascal. Program 1 had almost no parameters and all its interprocedural information changes were made by global variables. Both algorithms found program 1 to be very unstable, the correlation between the two algorithm's results for this program were
Table 2.1: Table comparing speed and accuracy of logical stability computation from [YC84]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>684</td>
<td>21</td>
<td>5312</td>
<td>263</td>
<td>5</td>
<td>0.94</td>
</tr>
<tr>
<td>2</td>
<td>1127</td>
<td>23</td>
<td>8150</td>
<td>119</td>
<td>1.5</td>
<td>0.61</td>
</tr>
<tr>
<td>3</td>
<td>487</td>
<td>15</td>
<td>1059</td>
<td>34</td>
<td>3.2</td>
<td>0.73</td>
</tr>
<tr>
<td>4</td>
<td>1744</td>
<td>46</td>
<td>44897</td>
<td>1318</td>
<td>2.9</td>
<td>0.88</td>
</tr>
<tr>
<td>5</td>
<td>1735</td>
<td>30</td>
<td>9318</td>
<td>551</td>
<td>5.9</td>
<td>0.58</td>
</tr>
<tr>
<td>6</td>
<td>1115</td>
<td>31</td>
<td>8724</td>
<td>54</td>
<td>0.6</td>
<td>0.16</td>
</tr>
<tr>
<td>All</td>
<td>6892</td>
<td>168</td>
<td>77460</td>
<td>2339</td>
<td>3</td>
<td>0.75</td>
</tr>
</tbody>
</table>

high at 0.94. The correlation between the logical stability measures for program six was low 0.16. It could be concluded from this that the new algorithm's results only correlated well with the old algorithm's results when the program in question was very unstable and had a lot of global variables. This could possibly be a problem with the rejection of intramodular information when calculating logical stability. Yau and Chang's algorithm only looks at the information flow between global variables as there are no local variables at the design stage.

Two algorithms are presented in [Cha84] for computing ripple effect. The first, mentioned above and implemented as a tool, does not consider intramodular information. The second, which does consider intramodular information unfortunately is not implemented as a tool.

Our reformulation of the algorithm also rejects some intramodular information for reasons of computational speed. It falls between Hsieh's accurate algorithm for computation of ripple effect and Yau and Chang's design algorithm. Intramodular information is taken into account, but control flow is not.

Yau and Chang improved the situation regarding problems with computation time but only for a limited version of the logical stability measure. Their algorithm was much faster than Hsieh's but it seems that the logical stability results only correlated with the much slower, more accurate version if the program was not very stable i.e. had a lot of global variables and ripple effect. Their approach of getting feedback at design level meant that steps could
be taken to make programs more stable or highlight specific problems from an early stage. But, there is a tradeoff in that the information gained was not as accurate as information derived from code level measurement.

2.4.2 Tools Which Trace Ripple Effect Through a System

Joiner et al used ripple effect analysis along with dependence analysis and program slicing to produce DPUTE [JT93], a Data-centered Program Understanding Tool Environment. DPUTE can be used during software maintenance of COBOL systems to enhance program understanding and to facilitate restructuring and reengineering of programs. Program slicing [Wei84] is used to compute intramodule change propagation. They found that otherwise ripple effect analysis could only be semi-automatic [Jet al93]. DPUTE uses a browser to highlight variables whose path can then be traced via forward or backward slicing. Different levels of ripples are shown in different colours so that users can distinguish them. The variable name, dependence type and line number in the source file are all displayed.

Problems were encountered during the automation of the intramodule change propagation stage of ripple effect analysis so a generalised program slicing technique was used to reduce the size of slices. Ripples were classified by Wang [Wet al96] into two categories:

1. Direct ripples - those introduced directly by the initial change

2. Induced ripples - those caused by direct ripple or other induced ripples.

Wang et al found that the average size of a potential ripple (including both direct and induced ripples) across a program could contain as much as 32% of the source code. Direct ripples affected only 1.5% of source code, thus concentrating on direct ripples was much more manageable. DPUTE only considers direct ripples.
SEMIT [CW87] shown in Figure 2.3, is a ripple effect analysis tool which is based on both semantic and syntactic information. It creates a syntax and semantics database for software which directly links the program's semantic information with its syntax. The syntax analysis program builds an initial semantic database based on program control flow and data flow. For each procedure within a program all external data used and modified by the procedure are represented. Syntactic and semantic information is linked by by grouping relations into dependencies based on 'modifies-uses paths'. All possible ripple effect paths are identified by SEMIT, interaction with an expert maintainer is then needed to define which are the more probable paths. The aim of SEMIT is to provide maintainers with up-to-date semantic information directly linked to the source code under observation and then express the meaning of that code, thus improving program understanding.

2.5 Other Work

In [TM94], Turver and Munro survey existing ripple effect analysis techniques. They find that a weakness with existing ripple effect techniques is that they can not be applied in the earlier stages of the software lifecycle. To address this weakness they use Ripple Propagation Graphs (RPG) to model the hierarchical structure of system documentation with the aim of measuring the impact of a change on the entire system. A logical model of documentation is created with the relationship consists-of linking parts of the documentation to each other, for example, Chapter Entity consists-of one or more Section Entities. This information is
then modelled in a RPG, and RPG crystallization performed to determine the constituent entities of the documentation [TM94, p.45]:

- Create the hierarchical graph structure of the document.

- Define a set of application themes (data objects in the documentation).

- Analyse each segment entity for themes and for each theme found, create a theme vertex and attach it to the segment vertex. This records the thematic dependencies.

- Connect together each co-occurrence of themes.

Set notation is used to represent all possible connections between documentation entities, and logical ripple effect analysis used to determine all parts of a document affected by a change. A probability connection matrix using techniques described in [Han72] and [Soo77] is also used to produce the probable maximum ripple effect. The rationale being that past recorded experience can be included in the probability matrix thus giving the maintenance manager more accurate information about their system.

Cantora et al [CLT96] propose a method to track the side effects of a maintenance operation to code by analysing potential and actual relationships. Relationships existing in code are split into potential: exist where unit $x$ may refer to any component of unit $y$, and actual relationships which exist where the code of unit $x$ contains direct or indirect reference to some units of component $y$. Actual relationships are a subset of potential relationships and any given maintenance operation can transform a potential relationship into an actual relationship. The method is based upon the definition, use and computation of Boolean matrices. It traces ripple effect from a given point in a program and outputs a list of variables which would be affected by an initial change due to maintenance.
Several algorithms for calculating the ripple effect are presented in [YL88], they are presented as not suitable for practical use, thus are for theoretical use only. The algorithms provide ripple effect calculation for sub-sections of ripple effect analysis computation e.g. intermodule propagation.

2.6 Summary

This chapter has introduced and explained the fundamental contribution of Yau and Collofello's work to the measurement of ripple effect and logical stability. Ripple effect has been identified as useful as part of the software maintenance process, we have looked at several software maintenance models which include ripple effect as one of their phases. Contributions to research into ripple effect have been looked at, in particular those that have produced tools to compute ripple effect automatically.
Chapter 3

Ripple Effect as a Matrix Product

The purpose of this chapter is to give a precise definition of the calculation of ripple effect and logical stability. Two fundamental ideas in the computation of ripple effect are: intramodule and intermodule propagation. This chapter gives a detailed description of what they are and the way in which they are calculated. This is followed by the computation of ripple effect for an example program to clarify the use of the ripple effect algorithm in practice. Central to this chapter is a reinterpretation in terms of matrix arithmetic, of what Yau and Collofello meant in their original algorithm. Initially, this exercise was intended as a means of clarifying exactly what Yau and Collofello meant by their sometimes rather obscure instructions. However, the mathematics, as is often the case, highlighted the different actions of the algorithm leading to the simplification described in Chapter Four.

3.1 Intramodule Change Propagation

The computation of ripple effect is based on the effect that a change to a single variable will have on the rest of a program. Given the three lines of code contained in Module $m_1$ shown in Figure 3.1: a change to the value of $d$ in (1) will affect the value of $a$ in (1), which will propagate to $a$ in (2). In (2) $a$ will affect $d$ which will then propagate to $d$ in (3). Propagation of change from one line of code to another within a module is called
intramodule change propagation. Starting points for intramodule change propagation such as \( a \) in line (1) can be thought of as 'definitions', the variable is being defined or given a value. Propagation then emanates from the defined variable through the module across the module boundary and into other modules (intermodule change propagation).

Yau and Collofello treat 'definitions' as any of the following occurrences. Code snippets given here with corresponding line numbers to illustrate the definitions are taken from Example.c, listed in Figure 3.2. Example.c is a simple program containing five functions which accepts any number of integers input by the user, performs several trivial calculations and then outputs the results.

1. The variable is defined in an assignment statement. For example caverage:

   \[
   42 \quad \text{caverage} = \frac{\text{total}}{\text{ccounter}};
   \]

2. The variable is assigned a value which is read as input. For example &vnumber:

   \[
   28 \quad \text{scanf("%f", &vnumber)};
   \]

3. The variable is an input parameter to module \( m \). For example ccounter:

   \[
   38 \quad \text{float calcmean(float ccounter)}
   \]

4. The variable is an output parameter from a called module. For example counter:
counter = values();

5. The variable is a global variable. For example total:

caverage = total/counter;

Intuitively only globals on the right hand side of assignments should count. Any variable occurrence on the left hand side is receiving a value from the variable on the right hand side of the assignment, thus whether it is global or not is irrelevant. In this instance if caverage is global this specific occurrence is not going to affect anything. A 0-1 vector $V_m$ can be used to represent the variable definitions in module $m$. Variable occurrences which satisfy any of the above conditions are denoted by '1' and those which do not by '0'. We shall use the notation $x_i^d$ ($x_i^u$) to denote a definition (use) of variable $x$ at line $i$. For example, $a_1^d$ means variable $a$ is defined in line 1 and $a_2^u$ means variable $a$ is used in line 2. Vector $V_m$ for the code in our example in Figure 3.1 (where $a$ is assumed global) is therefore:

$$V_{m1} = \begin{pmatrix} a_1^d & d_1^u & d_2^d & a_2^u & d_3^u \\ 1 & 0 & 1 & 1 & 0 \end{pmatrix}$$

To show how Yau and Collofello's algorithm relates to our approximated algorithm an element by element comparison is given in Appendix C.

A 0-1 matrix $Z_m$ can be produced to show which variables' values will propagate to other variables within module $m$. The rows and columns of $Z_m$ represent each individual occurrence of a variable. Propagation is shown from row $i$ to column $j$. For example the propagation from $a$ in line 2 to $d$ in line 2 is shown at row 4 column 3 and not at row 3 column 4. For the above code we get the following matrix:
```c
#include<stdio.h>
#include<math.h>

float total;

float values();
float calcmean(float);
float calcpower();
void output(float,float);

main()
{
    float average, power, counter;
    counter = values();
    average = calcmean(counter);
    power = calcpower();
    output(average, power);
    return 0;
}

float values()
{
    float vnumber;
    float vcounter = 0;
    printf("Enter a value or -1 to calculate mean: ");
    scanf("%f", &vnumber);
    for (; vnumber != -1; vcounter = vcounter + 1)
    {
        total = total + vnumber;
        printf("Enter a value or -1 to calculate mean: ");
        scanf("%f", &vnumber);
    }
    return(vcounter);
}

float calcmean(float ccounter)
{
    float caverage;
    caverage = total / ccounter;
    return(caverage);
}

float calcpower()
{
    float xpower;
    xpower = pow(total, 3);
    return(xpower);
}

void output(float oaverage, float opower)
{
    printf("A = total of values entered: %7.2f \n", total);
    printf("B = mean of values entered: %7.2f \n", oaverage);
    printf("C = A cubed: %7.2f \n", opower);
}
```

Figure 3.2: Example.c
We observe that $Z_{m1}$ is reflexive and transitive; that is, every variable occurrence is assumed to propagate to itself, and if $v_1$ propagates to $v_2$ and $v_2$ propagates to $v_3$ then $v_1$ also propagates to $v_3$. $Z_m$ therefore represents the transitive closure of variables within module $m$. In graph theory terms we conclude that $Z_{m1}$ represents the reachability matrix of some directed graph. This fact will be crucial to the simplification which we describe in Chapter 4.

### 3.2 Intermodule Change Propagation

Propagation from one module to another is called *intermodule change propagation*. A change to a variable can propagate to other modules if (code snippets are again taken from `Example.c` which is listed in Figure 3.2):

1. The variable is a global variable. For example `total`:

   ```
   42   caverage = total/ccounter;
   ```

   Intuitively only globals on the left hand side of an assignment should count - this specific occurrence of `total` is not propagating to anywhere, but we are sticking to Yau and Collofello's definition.

2. The variable is an input parameter to a called module. For example `counter`: 

\[
Z_{m1} = \begin{pmatrix}
  a_1^d & d_1^d & d_2^d & a_2^u & d_3^u \\
  1 & 0 & 1 & 1 & 1 \\
  d_1^u & 1 & 1 & 1 & 1 \\
  d_2^u & 0 & 0 & 1 & 0 & 1 \\
  a_2^u & 0 & 0 & 1 & 1 \\
  d_3^u & 0 & 0 & 0 & 0 & 1
\end{pmatrix}
\]
average = calcmean(counter);

3. The variable is an output parameter of module $m$. For example $vcounter$:

    return($vcounter$);

If the code in Figure 3.1 is part of module $m_1$ then $d$ clearly propagates to any module calling $m_1$. If $a$ is global then its occurrence on the left-hand-side of the assignment in line (1) will cause propagation to any modules using $a$. Suppose that the code constituting module $m_1$ is called by a module $m_2$, that $a$ is global and module $m_2$ uses $a$ and that a further module $m_3$ uses $a$ and $d$. The $(i,j)$th entry is 1 iff variable $i$ propagates to module $j$. We can represent the propagation of these variables using a further 0-1 matrix $X_{m1}$:

$$
X_{m1} = \begin{pmatrix}
    a^1 & 0 & 1 & 1 \\
    d^1 & 0 & 0 & 0 \\
    d^2 & 0 & 0 & 0 \\
    a^2 & 0 & 0 & 0 \\
    d^3 & 0 & 0 & 1
\end{pmatrix}
$$

Note that there is no propagation from any variable occurrence to $m_1$ i.e. column 1 is all zeros, because intermodule change propagation involves flow of program change across a module boundary (see Appendix B, Step 4).

Our matrix formulation of Yau and Collofello's algorithm continues by specifying various matrix products. We make choices between using the Boolean product ($V \cdot \land$) and the standard product ($+ \cdot \times$). This is done to capture, as closely as possible, what Yau and Collofello appear to be saying in their original algorithm (Appendix B). However we will
write the products out in the usual notation \((A \times B = AB)\) both to avoid excessive notation and to emphasise the fact that other choices are possible for each product (indeed we could follow [CG79] and define a whole 'minimax algebra' of different versions of Yau and Collofello's algorithm.

We now observe that the intermodule change propagation of all variable occurrences in \(m\) can be found by finding the Boolean product of \(Z_{m1}\) and \(X_{m1}\) giving:

\[
Z_{m1}X_{m1} = \begin{pmatrix}
1 & 0 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 \\
0 & 0 & 1 & 0 & 1 \\
0 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
0 & 1 & 1 \\
0 & 0 & 0 \\
0 & 0 & 0 \\
0 & 0 & 1 \\
0 & 0 & 1
\end{pmatrix}
= \begin{pmatrix}
0 & 1 & 1 \\
0 & 1 & 1 \\
0 & 0 & 1 \\
0 & 0 & 1 \\
0 & 0 & 1
\end{pmatrix}
\]

We use the Boolean product of \(Z_{m1}\) and \(X_{m1}\) to maintain consistency with intramodule change propagation computation which is also Boolean. The standard matrix product of \(V_{m1}\) and \(Z_{m1}X_{m1}\) shows propagation to each module from variable occurrences in module \(m\):

\[
V_{m1}Z_{m1}X_{m1} = \begin{pmatrix}
1 & 0 & 1 & 1 & 0
\end{pmatrix}
\begin{pmatrix}
0 & 1 & 1 \\
0 & 1 & 1 \\
0 & 0 & 1 \\
0 & 0 & 1 \\
0 & 0 & 1
\end{pmatrix}
= \begin{pmatrix}
0 & 1 & 3
\end{pmatrix}
\]

In this instance we can see from matrix \(V_{m1}Z_{m1}X_{m1}\) there are 0 propagations to module \(m_1\), 1 to module \(m_2\) and 3 to \(m_3\).
3.3 Complexity and Logical Stability

A complexity measure is factored into the computation by Yau and Collofello so that the complexity of modification of a variable definition is taken into account. Matrix C represents McCabe's cyclomatic complexity [McC76] for the modules in our code, shown in Figure 3.1 (the values for $m_2$ and $m_3$ have been chosen at random):

$$C = \begin{pmatrix} m_1 & 1 \\ m_2 & 1 \\ m_3 & 1 \end{pmatrix}$$

The product of $V_{m1}Z_{m1}X_{m1}$ and C is:

$$V_{m1}Z_{m1}X_{m1}C = \begin{pmatrix} 0 & 1 & 3 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = 4$$

This number represents the complexity-weighted total variable definition propagation for module $m$. If we now multiply this by the reciprocal of the number of variable definitions in module $m$: $\frac{1}{|V_m|}$ we get the mean complexity-weighted variable definition propagation per variable definition in module $m$. In our example $|V_m| = 3$ (no. of 1s in $V_{m1}$), the ripple effect for module $m$ is defined to be:

$$\frac{4}{3} = 1.33$$

The logical stability measure for module $m$ is defined to be its reciprocal:

$$\frac{3}{4} = 0.75$$

No program with a Ripple Effect of zero was found in this study, but if there were we could adopt a definition of Logical Stability being $\frac{1}{1 + RE}$.

In the next section we present an example based on actual code.
3.4 Computing Ripple Effect for an Example Program

The following is an example of the computation of logical stability of a program called Example.c. A listing of the program source code is given in 3.2. The following algorithm can be used to calculate ripple effect and logical stability. It is split into eleven separate steps, steps one to nine should be followed iteratively for each module, steps ten and eleven for the program as a whole.

**STEP 1** — Make a list of all occurrences of variables (not including variable declarations) for each module in the order in which they appear in the source code. For example, in function *values* the list would be:

$vcounter, vnumber, vnumber, vnumber, vcounter, vcounter, total, total, vnumber, vnumber, vcounter$

**STEP 2** — Form Boolean vectors for each module. We give $V_{values}$ as an example, with '1' in the vector denoting that the variable occurrence satisfies one or more of the five conditions specified in section 3.1, and '0' that it satisfies none. By inspection of the source code in 3.2:

$$
V_{values} = \begin{pmatrix}
vc_2^d & vn_2^d & vn_2^u & vc_2^d & vc_2^u & tot_3^d & tot_3^u & vn_3^u & vn_3^d \\
1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0
\end{pmatrix}
$$

**STEP 3** — Construct matrix $Z_m$ for each module:
STEP 4 — Construct matrix $X_{values}$:

$$Z_{values} = \begin{pmatrix}
  v_{c25}^d & v_{n28}^d & v_{n29}^u & v_{c29}^d & v_{c29}^u & t_{o31}^d & t_{v31}^u & v_{n33}^d & v_{c35}^u \\
  1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\
  0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
  0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\
  0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\
  0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
\end{pmatrix}$$

$$X_{values} = \begin{pmatrix}
  v_{c25}^d & v_{n28}^d & v_{n29}^d & v_{c29}^d & v_{c29}^u & t_{o31}^d & t_{v31}^u & v_{n33}^d & v_{c35}^u \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{pmatrix}$$

$\text{main values calcmean calcpower output}$
STEP 5 — Construct matrix $C$ representing the McCabe complexities for the modules in this program which are as follows:

$$
\begin{array}{c}
\text{main} \\
\text{values} \\
\text{calcmean} \\
\text{calcpower} \\
\text{output}
\end{array} = 
\begin{pmatrix}
1 \\
2 \\
1 \\
1 \\
1
\end{pmatrix}
$$

STEP 6 — Find the Boolean product of the matrices formed in STEP 3 and STEP 4, $Z_{values}X_{values}$:

$$
\begin{pmatrix}
1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\
0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0
\end{pmatrix}
= 
\begin{pmatrix}
1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 \\
0 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0
\end{pmatrix}
$$

STEP 7 — Find the product of the matrices formed in STEP 2 and STEP 6, $V_{values}Z_{values}X_{values}$:
\[
\begin{pmatrix}
1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0
\end{pmatrix}
\begin{pmatrix}
1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 \\
0 & 0 & 1 & 1 & 1 \\
0 & 0 & 1 & 1 & 1 \\
1 & 0 & 0 & 0 & 0 \\
\end{pmatrix}
= \begin{pmatrix}
2 & 0 & 4 & 4 & 4
\end{pmatrix}
\]

**STEP 8** — Find the product of the matrices formed in **STEP 2** and **STEP 5**, \(V_{value}, Z_{value}, X_{values}:

\[
\begin{pmatrix}
2 & 0 & 4 & 4 & 4 \\
\end{pmatrix}
\begin{pmatrix}
1 \\
2 \\
1 \\
1 \\
\end{pmatrix}
= 14
\]

**STEP 9** — Divide this figure by \(|V_m|\), the number of variable definitions in module \(m\) to give the measure of ripple effect for the module. Looking at \(V_{value}\) we can see that there are 6 '1's in the vector, we therefore divide the product of the matrices by 6 giving:

\[
\frac{14}{6} = 2.33
\]

the Ripple Effect values for the other modules are:

\(RE_{\text{main}} = 1.33\)

\(RE_{\text{calcmean}} = 1\)

\(RE_{\text{calcpower}} = 1\)

\(RE_{\text{output}} = 0\)
The Logical Stability for module values is therefore $\frac{6}{14} = 0.43$ and for the other modules in program example.c:

\[
LS_{\text{main}} = 0.75
\]
\[
LS_{\text{calcmean}} = 1
\]
\[
LS_{\text{calcpower}} = 1
\]
\[
LS_{\text{output}} = 0
\]

**STEP 10** — Calculate the Ripple Effect for the Program as a whole using:

\[
REP = \frac{1}{n} \sum_{i=1}^{n} \frac{V_{mi}Z_{mi}X_{mi}.C}{|V_{mi}|}
\]  \hspace{1cm} (3.4.1)

Where $m = \text{module and } n = \text{number of modules.}$

Therefore $REP_{\text{example.c}} = \frac{34}{30} = 1.13$

**STEP 11** — Finally the Logical Stability measure for the Program Example.c is the reciprocal ripple effect measure:

\[
LSP = \frac{1}{REP} = \frac{30}{34} = 0.88
\]

### 3.5 Summary

In this chapter the simplified algorithm for computing logical stability has been explained. Intramodule and intermodule change propagation have been introduced and discussed in detail with example code snippets given to clarify their meaning. Finally the computation of ripple effect and logical stability for an example program has been shown.
Chapter 4

Deriving an Approximation

In this chapter we show how matrix \( Z_m \) is produced automatically using information held in two matrices, \( D_m \) and \( A_m \). Matrix \( D_m \) holds information on definition/use and matrix \( A_m \) holds information on assignment pairings within the code. Due to computation constraints we choose to compute an approximated version of \( D_m \), \( D'_m \) which does not consider control flow information. Previous attempts at automating ripple effect measurement have suffered from slow computation times (see section 2.4). It is with this problem in view that approximated \( D_m \) is introduced and we give a description of the potential problems that ensue. Section 4.5 discusses matrix \( C \) the complexity matrix. To keep the ripple effect measure aligned with Yau and Collofello's original ideas ripple effect has been computed using four versions of matrix \( C \); an explanation is given of each of these and the reasons behind their inclusion.

4.1 Decomposition of Matrix \( Z_m \)

Matrix \( Z_m \) represents intramodule change propagation. This is clearly a transitive relation because if change propagates from variable occurrence \( v_i \) to \( v_j \), also from \( v_j \) to \( v_k \), then a change in occurrence \( v_i \) will propagate to \( v_k \) via \( v_j \). Thus, as we observed earlier, \( Z_m \) is the matrix of a transitive relation and represents the reachability matrix of some basic relation.
To determine $B_m$ is not difficult: change propagates either from the right-hand side of an assignment to the left-hand side, or it propagates from the definition of a variable to a subsequent use of that same variable. These two modes of propagation are referred to as 'assignment' and 'definition/use', respectively. If we treat the two as different relations, represented by matrices $A_m$ and $D_m$, respectively, then we see that $B_m = A_m + D_m$.

To explain this we turn to Figure 4.1 where information flow from one variable occurrence to another is shown using arrows, variable occurrence $x$ takes its value from $y$ in line 1, thus $x,y$ is an assignment pair. Information about such pairings is held in matrix $A_m$. The definition of $x$ in line 1 is used by $x$ in line 2. This is a definition/use association. Information about definition/use associations is held in matrix $D_m$.

The combination of information from assignment and definition/use gives us information about the flow of values from one variable to another within a module. From this information we can work out which variables would be affected if we changed any particular variable occurrence. The assignment matrix $A_m$ which holds information about all assignment pairings for our example code is as follows:

$$A_m = \begin{pmatrix}
x_1^d & y_1^u & y_2^d & x_2^u \\
x_1^d & 0 & 0 & 0 \\
y_1^u & 1 & 0 & 0 \\
y_2^d & 0 & 0 & 0 \\
x_2^u & 0 & 0 & 1 \\
\end{pmatrix}$$
The definition/use association matrix $D_m$ is as follows:

$$D_m = \begin{pmatrix}
  x_1^d & y_1^u & y_2^d & x_2^u \\
  x_1^d & 0 & 0 & 0 & 1 \\
  y_1^u & 0 & 0 & 0 \\
  y_2^d & 0 & 0 & 0 \\
  x_2^u & 0 & 0 & 0 \\
\end{pmatrix}$$

Matrix $A_m$ and matrix $D_m$ have all variable occurrences as rows and columns, even though in $A_m$ only defined variables are needed as columns and in $D_m$ only defined variables are needed as rows. The sum of these matrices then give us matrix $B_m$ representing direct intramodule change propagation. The information now held in matrix $B_m$ is also shown in Figure 4.2.

$$B_m = \begin{pmatrix}
  x_1^d & y_1^u & y_2^d & x_2^u \\
  x_1^d & 0 & 0 & 0 & 1 \\
  y_1^u & 1 & 0 & 0 & 0 \\
  y_2^d & 0 & 0 & 0 \\
  x_2^u & 0 & 0 & 1 & 0 \\
\end{pmatrix}$$
We can now find the reachability matrix (equivalent to the transitive closure) for \( B_m \), namely \( Z_m \), using:

\[
Z_m = I \lor B \lor B^2 \lor \ldots \lor B^n
\]

\( n \) = number of variable occurrences, in this case four.

The reachability matrix shows all possible links between any variable occurrence and any other variable occurrence within the module. From the information now contained within matrix \( Z \) any change to a variable occurrence can be tracked throughout the module and the ramifications of its change calculated.

\[
\begin{pmatrix}
  x_1^d & y_1^u & y_1^d & x_2^u \\
  x_1^l & 1 & 0 & 1 & 1 \\
  y_1^u & 1 & 1 & 1 & 1 \\
  y_1^d & 0 & 0 & 1 & 0 \\
  x_2^u & 0 & 0 & 1 & 1
\end{pmatrix}
\]

4.2 Approximated Matrix D

To produce matrix \( D_m \) (the definition/use matrix) control-flow information has to be taken into account. The three basic control flow constructs are:

- sequence
- selection or branching
- iteration or looping
The code fragment used so far in this chapter has contained only sequential code, we also need to look at code which contains loops and branches and how this affects the accuracy of $D_m$. Replacing $D_m$ with a matrix of all possible definition/use pairs greatly simplifies the computation, we name this matrix $D'_m$. By defining $D'_m$ to ignore control flow information we are essentially treating the code as if it were contained within a loop, except in the case of one definition being killed by another (further details are on page 55). This means that the more code within a module that is contained within loops the closer the match between $D_m$ and $D'_m$. If we include all possible definition/use pairs matrix $D'_m$ for the example code in Figure 4.1 will become:

$$
D'_m = \begin{pmatrix}
    x_1^d & y_1^u & y_2^d & x_2^u \\
    x_1^d & 0 & 0 & 0 & 1 \\
    y_1^u & 0 & 0 & 0 & 0 \\
    y_2^d & 0 & 1 & 0 & 0 \\
    x_2^u & 0 & 0 & 0 & 0 
\end{pmatrix}
$$
Figure 4.4: False definition/use pairing in matrix $D'$

Note that there is one false entry in matrix $D'_m$: $y^d_2$ is paired with $y^n_1$, see figure 4.4. Note also that matrix $D'_m$ for each module in Figure 4.3 will be identical for all the code fragment examples as control flow is not taken into account.

Both $D_m$ (below) and $D'_m$ for the code contained in a loop are identical, i.e.

$$D_m = D'_m$$

This is because all definition/use pairings are true in both cases.

$$D_m = \begin{pmatrix} x_1^d & y_1^n & y_2^d & x_2^n \\ x_1^d & 0 & 0 & 0 & 1 \\ y_1^n & 0 & 0 & 0 & 0 \\ y_2^d & 0 & 1 & 0 & 0 \\ x_2^n & 0 & 0 & 0 & 0 \end{pmatrix}$$

Matrix $D_m$ for code which contains selection or branching is affected as follows. Consider the same source code example used above but now containing selection (see figure 4.3). Note that the if statement does not contain a variable and both if and end if statements are not given line numbers to enable easy comparison with previous matrices. Both lines 1 and 2 cannot be true at the same time. $D_m$ for this version of the code is:
We also need to look at sequential code where the definition of a variable occurrence is followed by another definition of that variable, see Figure 4.5. The value of \( z \) in line 1 will be killed when control flow reaches line 2 and \( z \) is assigned the value 2. On examination of the programs used in our study, detailed in Chapter 6, we find that killed definitions do not occur. As such we do not consider them further in this thesis.

As matrix \( D' \) includes all definition/use pairs regardless of control flow there is an extra '1' highlighted below at row 1, column 5.
We have seen in this section that the use of matrix $D_m'$ in computing ripple effect can cause false relations when dealing with sequential and selective code. In general if $F_I$ (iterative), $F_S$ (sequential) and $F_B$ (branching) are identical code fragments except that $F_I$ contains some loops and $F_B$ some branching and $\Phi(F) =$ number of 1s in $D_m$ then:

$$\Phi F_I \geq \Phi F_S \geq \Phi F_B$$

Intuitively, our approximation will therefore be least accurate when there are many branches and no loops. The next section discusses Matrix C: we present several variations of McCabe’s complexity in an attempt to counteract this effect.

### 4.3 Matrix C

Using McCabe’s measure via matrix C introduces complexity into the computation of ripple effect. Yau and Collofello include McCabe’s cyclomatic complexity as a part of their algorithm. In our reformulation we have thus far used the cyclomatic complexity measure for each module as the rows in matrix C:
Shepperd [She88] criticised cyclomatic complexity for being based on poor theoretical foundation and being outperformed as a measure of general complexity by lines of code (see section 1.4). Our concern is somewhat different because we expect that our approximation will be least accurate when there are many branches and few loops (see previous section). Thus we have a different reason for preferring a variant of McCabe. As mentioned previously McCabe's cyclomatic complexity measure is a count of all conditions within the module plus one. We present four versions of matrix $C$ in response to the question of accuracy of matrix $D_m'$.

### 4.4 Versions of the McCabe Complexity Factor

In Yau and Collofello's [YC80] ripple effect algorithm McCabe's cyclomatic complexity is used: "...[to] provide more realistic measures of the amount of effort required to analyze the program to ensure that inconsistencies are not introduced" [YC80, p. 548]. By introducing several variations of McCabe's complexity measure we are trying to show that we can counteract the effect that our approximation of matrix $D_m$ will have on the final ripple effect measure. Points to consider are: code contained in a loop is accurately represented in matrix $D_m'$, sequential code is less accurately represented than code within a loop and code containing branching is least accurately represented. We were interested to find that the
highest correlation with Yau and Collofello's original ripple effect in the sample programs was found using the Original version of McCabe. The four versions of the complexity factor are as follows:

1. Control McCabe - matrix C contains the value 1 for each module. The control McCabe version of matrix C when multiplied with the other matrices will produce a measure which has no complexity element factored in. This means it can give us a baseline figure for the ripple effect of any particular module or program.

2. Original McCabe - all conditions plus one.

3. Loops McCabe - number of loops plus one, branches are not counted. To counteract the effect that $D'_m$ will have on the ripple effect measure we could consider reducing the original version of matrix C according to how much sequence and selection is present in the target code. This would introduce further complexity into the computation and not really be in keeping with the ethos of cyclomatic complexity. Therefore instead of reducing the complexity factor for sequential and selective code we decided to count only loops and not branches.

4. Branches McCabe - number of branches plus one, loops are not counted. Similar to the above version but counting branches only and not loops.

4.5 Summary

In this chapter the intramodule change propagation matrix $Z_m$ has been introduced and a description given of how it is produced from matrices $A_m$ and $D_m$. The approximated version of matrix $D_m$, namely matrix $D'_m$, has been described with examples illustrating the effect that control flow has on its accuracy. Matrix $D'_m$ is more accurate when code
contains loops, less accurate if it contains only sequential or selective code; reasons for this are given. Matrix $C$ introduces a complexity factor to the algorithm and as such is shown in several variations which may counterbalance the inaccuracy of $D'_m$ and thus produce an optimum ripple effect measure. An analysis of the effect of the different versions of McCabe is given in Chapter 6.
Chapter 5

Ripple Effect and Stability Tool

This chapter is a description of REST (Ripple Effect and Stability Tool) which has been built at the Centre for Systems and Software Engineering at South Bank University. The need for the tool, its place in relation to other ripple effect measurement tools, and its contribution towards our research are explained. The individual components making up REST are detailed with a description of how they all link together to produce ripple effect measures.

5.1 The Need for Fully Automated Computation of Ripple Effect

A tool has been produced which computes ripple effect measures automatically but which uses an approximation of intramodule change propagation described in Chapter 4. Several tools can track ripple effect through a system but can only do this semi-automatically: there needs to be some user intervention at some point to make decisions which the tool cannot make. Our tool produces fully automatic, approximated ripple effect measures. Automation of ripple effect can focus in one of two directions: the computation of ripple effect measure or the tracing of ripple effect on variables through a program or system. Tools have been produced for both of these categories of ripple effect. The focus of our
work is on computing the measure rather than tracing affected variables, and as such our
tool can be directly compared with tools in section 2.4.1. REST is one of the few tools
which specifically gives a ripple effect measure of a program. How much use this is to
maintainers of source code will be the subject of future research. Our product-process case
study makes a start in this direction.

5.2 The REST Software Tool

REST was built at the Centre for Systems and Software Engineering as part of a project
partially funded by British Telecommunications (BT) laboratories. REST comprises four
separate software modules: a Parser, Listfuns, Funmat and Ripple, as detailed in Figure
5.1. Its aim is to produce ripple effect measures for C and C++ code as an addition
to BT's comprehensive suite of measurement tools the Code Measurement Toolkit (CMT).
The CMT [HL97] is an integrated environment for the code analysis and maintainability
assessment of C and COBOL code. It was developed after BT carried out an analysis of
their software encompassing five large software projects within BT. The results indicated
that it should be possible to predict with 70-80% accuracy which source code files in a
system are likely to require changing in the future. The CMT also uses X-RAY [BW99]
(described below) and QUALMS [BL90] both developed at the Centre for Systems and
Software Engineering.

5.2.1 Parser

Initial attempts at parsing source code were made using Unravel [LWG+95], a prototype
static analysis tool. Unravel scans C source code, parses it and translates it into a lan-
guage independent format. An interactive slicing component can then be used to produce
backward and forward slices through the code. Unfortunately we were unable to use Unravel to produce the required input for the rest of the ripple software because it did not contain enough information for our purposes. A second attempt at parsing source code for our tool was made using information produced by the X-RAY tool. X-RAY is a maintenance tool [BW99] which does extensive cross referencing of variables. It contains a general purpose parser which was designed for COBOL and FORTRAN, unfortunately it proved insufficiently flexible for C and C++. X-Ray is driven by a language definition file for each language to be parsed. There are actually two independent parsing mechanisms in use by X-Ray, which both use top-down BNF type language definitions. The original parser has a look-ahead of only one symbol, a number of mechanisms are used to overcome this basic limitation so that most earlier languages can be parsed successfully. However, with C++ a more comprehensive compiler with an extended look-ahead capability was required. LEX
and YACC were also considered but are not flexible enough for C++.

After reviewing what was available at the time we decided to use PCCTS (Purdue Compiler Construction Tool Set) [Par96] from Purdue University, Indiana, USA, which was freely available at that time in Version 1.33 written in C++. There is now an improved version, 2.7, written in Java which is also free [par01]. The parser was written by Lasitha Leelasena, and has been adapted by David Wigg and Sue Black, all from the CSSE at South Bank University.

The PCCTS parser has been adapted to produce the necessary information for input to Listfuns and Funmat. All the parser output files and their contents for the example module m1 (Figure 5.3) are detailed in Figure 5.2. Fun2.v.dat contains the information needed to produce the V matrix. The first column contains information about the variable occurrence: whether it is a use - 0, a definition - 1, or both - 2 (this is changed to a 1 by Funmat for the V matrix). The next column shows the parser line number which has been invoked by the variable occurrence. The third column of information contains the variable occurrence's line number in the .i file. The .i file is the preprocessed version of the original C or C++ source code file. The final column is a description of the variable occurrence as produced by the parser. From this information the V matrix can be produced.

Also output from the parser is fun2.d.dat. The data in this file was previously used to produce matrix D', currently fun2.v.dat is used to produce it. Data from file Fun2.n.dat shows assignment pairings within module m1. The first column shows the variable name, the second column the line no. that the variable occurrence appears on in the .i version of the program. The third column shows whether the variable occurrence is being assigned 'to' or 'from', so for example in line 974 variable occurrence 'd' value is assigned from variable occurrence 'a'. Note that the line number is high i.e. 974 because the .i file is a preprocessed version of the program file and as such contains all header and library files.
1 Line 974: globalvariable a on LHS of assignment
0 Line 974: variable d not member of V
1 Line 975: variable d on LHS of assignment
2 -2373 Line 975: Globalvariable a on RHS of assignment
0 Line 976: variable d not member of V

Parser output data to file fun1_v.dat

a 974
d 974
d 975
a 975
d 976

Parser output data to file fun1_d.dat

a 974 to
d 974 from
d 975 to
a 975 from

Parser output data to file fun1_a.dat

a 974 1 a
d 974 0 -
d 975 0 -
a 975 0 -
d 976 1 b

Parser output data to file fun1_x.dat

File Fun1 = function m1
File Fun2 = function m2
File Fun3 = function m3

Parser output data to file fun_list.lis

m1 1
m2 1
m3 1

Parser output data to files mat_c.dat, mat_cloop.dat, mat_cb.dat

Figure 5.2: REST - All files output by parser for example program
Figure 5.3: Intramodule and Intermodule change propagation

included in the program. File fun2.x.dat contains information that is used to produce intermodule change propagation information. Again, the first two columns are the variable occurrence and the line number. The third column denotes, using a 1 or 0, whether the variable occurrence is involved in intermodule change propagation. The fourth column shows which type of intermodule change propagation the variable occurrence is involved in (see previous section). In this case variable occurrence 'a' on line 974 is a 'global variable on the left-hand-side of an assignment, and 'd' on line 976 is an 'input parameter to a called module'. We can check this by looking at Figure 5.3.

File fun_list.lis simply links the actual name of the function with the name automatically produced by the parser for purposes of recognition. File Mat.c.dat contains the McCabe's cyclomatic complexity [McC76] for each module within the program, the first column is the function name and the second column the complexity measure for that module. The same output is produced by the parser in this instance for the files mat1.cloop.dat and mat1.cb.dat. These are, respectively, the files that contain the Loops McCabe and Branches McCabe measures. These two versions of the McCabe measure are discussed along with the Control and Original McCabe versions in detail in section 4.4.
5.2.2 Listfuns

Listfuns, Funmat and Ripple were all built using Microsoft Visual C++ version 6.0. Listfuns is essentially a housekeeping type program, its main remit is to create the files needed to store the ripple matrices, to make sure that information is in the correct format and to produce different versions of the C matrix. The files that it outputs, shown in Figure 5.2, are read as input to Funmat and Ripple.

5.2.3 Funmat

Funmat takes several files as input, it gets data line by line from fun*.v.dat and fun*.a.dat and reads it into several arrays. These arrays are then used to produce mat*.v.dat, mat*.a.dat, mat*.dat and mat*.x.dat. Mat*.v.dat is produced from the first column of the fun*.v.dat file with any occurrence of a 2 being replaced with a 1. Thus matrix \( V \) for \( m1 \) is:

\[
V_m = \begin{pmatrix}
  a_1^d & d_1^u & d_2^d & a_2^u & d_3^u \\
  1 & 0 & 1 & 1 & 0
\end{pmatrix}
\]

N.B. Matrices relevant to the discussion of files and their output are reproduced here for ease of reference.

Mat*.a.dat is produced by reading in data from fun*.a.dat into two tables which are then cross read to match the assignment 'to' and assignment 'from' information thus producing all assignment pairings. Pairings are represented in mat*.a.dat by 1s, no pairing is represented by 0. Mat*.a.dat for our example code is:
Mat*.d.dat is produced by reading in data from fun*.v.dat into two tables which are cross read to match the 'definition' and 'use' information thus producing definition/use pairings. Pairings are represented in mat*.d.dat by 1s, no pairing is represented by 0. Mat*.d.dat for our example code is:

\[
A_m = \begin{pmatrix}
  a_1^d & d_1^n & a_2^d & a_3^n & d_3^n \\
  a_1^n & 0 & 0 & 0 & 0 \\
  d_1^d & 1 & 0 & 0 & 0 \\
  d_2^d & 0 & 0 & 0 & 0 \\
  a_2^n & 0 & 0 & 1 & 0 \\
  d_3^d & 0 & 0 & 0 & 0 
\end{pmatrix}
\]

\[
D'_m = \begin{pmatrix}
  a_1^d & d_1^n & a_2^d & a_3^n & d_3^n \\
  a_1^n & 0 & 0 & 0 & 0 \\
  d_1^d & 0 & 1 & 0 & 0 \\
  d_2^d & 0 & 0 & 0 & 0 \\
  a_3^n & 0 & 0 & 0 & 0 \\
  d_3^d & 0 & 0 & 0 & 0 
\end{pmatrix}
\]

Data from globals.lis and calls.lis are used to produce a call hierarchy matrix (call.dat) which is in turn used to compute whether variable occurrences are involved in intermodule change propagation. This information is needed to produce Matrix X which gets variable occurrence names from fun*.v.dat to produce the rows and function names from fun.list.lis to produce the columns. For the three intermodule change propagation categories information for category a - 'a global variable' comes from globals.lis. Information to produce category b - 'an output parameter' comes from calls.lis and category c - 'an input to a called module'
comes from function call data which is held in fun*.x3.dat. Matrix X is the most complicated to produce, most of Funmat is devoted to producing it:

$$X_m = \begin{pmatrix}
   m_1 & m_2 & m_3 \\
   a^d_1 & 0 & 1 & 1 \\
   d^u_1 & 0 & 0 & 0 \\
   a^d_2 & 0 & 0 & 0 \\
   d^u_3 & 0 & 0 & 1 
\end{pmatrix}$$

5.2.4 Ripple

The function of Ripple is to take the matrices output by Funmat and Listfuns and produce the ripple effect measure for each module in turn followed by the ripple effect for the program as a whole. Mat*.a.dat and mat*.d.dat are used to produce a reachability matrix for mat* which is written to file mat*.z.dat:

$$Z_m = \begin{pmatrix}
   a^d_1 & d^u_1 & d^d_2 & a^u_3 & d^u_3 \\
   a^d_1 & 1 & 1 & 1 & 1 \\
   d^u_1 & 1 & 1 & 1 & 1 \\
   a^d_2 & 1 & 1 & 1 & 1 \\
   d^u_3 & 0 & 0 & 0 & 1 
\end{pmatrix}$$

As full details of the manipulation of these matrices to produce ripple effect measures is given in chapter 3 we will not go into detail again here. Ripple effect measures are output to file output.dat.
5.3 Summary

In this chapter we have looked at the need for automatic computation of ripple effect measures. Several tools have been produced by research groups around the world, we have discussed where our tool fits in. The production of REST has greatly aided our research in that it facilitated fast and accurate production of ripple effect measures. A detailed description of REST and all its components has been given allowing the reader an insight into its workings at a low level. A small example program, propagation.i, has been used to illustrate the function and output of the REST components.
Chapter 6

Validation of the Approximation Algorithm

This chapter provides validation of the approximated version of the ripple effect measure by computing Yau and Collofello's ripple effect for several programs and then comparing this with the approximated ripple effect for the programs. Four versions of the C complexity matrix: Original, Control, Loops and Branches are used to compute differing versions of approximated ripple effect. The Pearson correlation coefficient is used to show the relationship between the Yau and Collofello ripple effect and the four versions of the approximated ripple effect. Fifteen programs in total were used in this experiment, eleven of the programs are discussed in this chapter, the remaining four programs are versions of a software tool and as such form part of a case study used in Chapter 7 to look at the product-process link. The eleven programs discussed in this chapter are divided into two groups: those programs where the Yau and Collofello ripple effect is identical to the approximated ripple effect, and those where it is not. Killed definitions (see page 55) are not discussed as they do not arise in any of the programs used in our study. Each group of programs is followed by a discussion of the results for that group. This chapter concludes with a discussion of spurious 1s and their effect on the approximated version of ripple effect and a summary of the main points discussed.
6.1 Programs with Identical Ripple Effect

Five of the fifteen programs under study produced exactly the same ripple effect for the approximated version as for the Yau and Collofello version. This section examines the programs and explains why they produce the same results. It begins with a summary of the findings for these programs and then goes on to discuss the programs in detail. The five programs under investigation in this section are: funmat5.c, conv.c, quadratic.c, genscrip.c, and tamilize.c. Funmat5.c is a file processing program in which most of the code within any given module is contained within a loop. Conv.c, tamilize.c and genscrip.c are string/character processing programs with many of their modules containing large loops also. The other modules contain several spurious 1s but because these 1s do not affect the variable occurrences which propagate via intermodule propagation there is no change in the ripple effect. Program quadratic.c contains no iteration, only sequence and selection. There are no spurious definition/use pairings and thus no spurious 1s.

6.1.1 Reasons for Identical Ripple Effect

Overall the reasons for these programs not differing in their ripple effect for the approximated version and the Yau and Collofello version seems to be the following factors:

1. All code within the module/program is within a loop, e.g.

```
loop
  1 a=b;
  2 b=c;
end loop
```

b is used in line 1 and then defined in line 2, as the code is within a loop b in line 2 can affect b in line 1. So Matrix $D'$ for this code would be correct.
2. Code is sequential and there are no variables used before being defined, e.g.

\[ \begin{align*}
1 & \quad a = c; \\
2 & \quad b = a;
\end{align*} \]

\( a \) is defined in line 1 before being used in line 2, thus there is no false definition/use pairing, so Matrix \( D' \) for this code would also be correct.

3. The spurious 1s represent variable occurrences which do not propagate to any of the interface variables, those variables through which propagation can affect other modules within the program.

Table 6.1 shows ripple effect measures for all the programs described in this section. Pearson's correlation coefficient for Yau and Collofello's ripple effect and the approximated ripple effect using the variants of the complexity factor (Section 4.3) are shown. Correlation with Number of modules and Lines Of Code are shown as a benchmark. If the correlation of either of these with Yau and Collofello's ripple effect were high, it would be easier to produce and use them instead. It can be seen from the table that correlation between Yau and Collofello's ripple effect and Original is 1. This is to be expected as the reason these programs have been grouped together is because the ripple effect is the same for the Original version. Correlation with Branches is high and Loops is low because these programs contain lots of selective code and not much iteration. Correlation with Control is negative.
1 void Get_D_info(const char *name, Data *table1,int *no_of_rows)
2 {
3    int i=0;
4    char c;
5    FILE *f = openfile(name,"r");
6    fscanf(f,"%s",&table1[i].var_name);
7    while((c = fgetc(f))!=EOF) 10
8           {
9              fscanf(f,"%s",&table1[i].line_no);
10             i++;
11             fscanf(f,"%s",&table1[i].var_name);
12            }
13     *no_of_rows = i;
14    fclose(f);
15 }

Figure 6.1: Module 4 of funmat5.c

which suggests that in computing ripple effect we do need to use a complexity factor of some sort. We now look at the individual programs in more detail.

6.1.2 Funmat5.c

Funmat5.c is a file processing program containing 245 LOC and 19 modules. It was written by the author as part of REST (Ripple Effect and Stability Tool). It performs mainly file processing, taking files output from a PCCTS produced C parser and reading the information contained in the files. It then takes out the information required to produce the matrices required for ripple effect computation and writes the matrices to file.

All modules have exactly the same ripple effect for the approximated version as for the Yau and Collofello version. In all but two of the modules within funmat5.c the code is contained within large loops, the other two modules being essentially trivial. As previously mentioned, if all code within a module is contained in a loop, the matrix $D'$ for the module will be exactly the same as the matrix $D$. Figure 6.1 is a typical module from funmat5.c. Almost all code within the module is contained within the while loop, any change to any variable
occurrences within the loop can affect any other variable occurrences. For example variable occurrence \( i \) in line 11 is affected by variable occurrence \( i \) in line 12 where it is incremented. If there was no \texttt{while} loop from line 9 to line 14 and thus the code was sequential \( i \) in line 11 could not be affected by \( i \) in line 12.

6.1.3 Conv.c

Conv.c is a string processing program containing 53 LOC and 3 modules. It was written by D.R. Kennard in 1992 as part of the TXEE telephone switching system produced by Nortel for British Telecom. The program reads in decimal and hexadecimal strings one character at a time from a file, checks that they are not comments and then outputs them to the screen. All modules have exactly the same ripple effect for the approximated version as for the Yau and Collofello version. The three modules are: \texttt{get.val}, \texttt{processbuf} and \texttt{main}. \texttt{Get.val} contains one large \texttt{if else} statement, which contains two \texttt{for} loops. All code within \texttt{processbuf} is contained within one large iteration which contains several smaller selection statements. \texttt{Main} is contained within one large \texttt{while} loop which contains a selection statement.

Matrix \( D' \) for module \texttt{get.val} is shown in Figure 6.2. As there are fifty-three variable occurrences in \texttt{get.val} there are fifty-three rows and columns in its matrix \( D' \). All definition/use pairs in the module are shown by a 1 in the matrix. Regions have been highlighted to show which areas of the matrix are affected by particular control statements. Two ‘for’ loops are highlighted, all 1s inside these regions are valid (not spurious) and three sections are shown that are affected by ‘if else’ statements, all 1s within these sections are spurious because the areas are mutually exclusive. Matrix \( D' \) for \texttt{get.val} contains 60 spurious 1s, but the ripple effect value remains unchanged. The majority of spurious 1s are false definition/use connections between the three parts of this selection. For example the variable occurrence \(*disp\) is being used in the following line of code:
Variables in 'for' loop - therefore can all affect each other
Variables in 'if else' statement - therefore cannot be affected

Figure 6.2: Matrix $D'$ for module `get_val` of conv.c
int get_val (buff, disp)
char *buff;
int *disp;
int temp = 0;
if ((buff[0] >= '0') && (buff[0] <= '9'))
{
    /* Decimal number */
    for (*disp = 0; (buff[*disp] >= '0') && (buff[*disp] <= '9'); (*disp)++)
    {
        temp = (temp * 10) + buff[*disp] - '0';
    }
}
else if ((buff[0] == 'h') || (buff[0] == 'H'))
{
    /* Hex number */
    for (*disp = 0; (buff[*disp] >= '0') && (buff[*disp] <= '9'); (*disp)++)
    {
        if ((buff[*disp] >= '0') && (buff[*disp] <= '9'))
        {
            temp = (temp * 16) + buff[*disp] - '0';
        }
        else if ((buff[*disp] >= 'A') && (buff[*disp] <= 'F'))
        {
            temp = (temp * 16) + buff[*disp] - 'A' + 10;
        }
        else temp = (temp * 16) + buff[*disp] - 'a' + 10;
    }
    else *disp = 0;
    return temp;
}
Figure 6.3: Module Get_val from conv.c
else if ((buff[*disp] >= 'A') && (buff[*disp] <= 'F'))

and is then defined later in the module:

else *disp = 0;

The second occurrence of *disp is mutually exclusive from the first due to the selective construct, but because all definitions are paired with all uses in matrix $D'$ all such occurrences are represented. There is no difference in the ripple effect measure even though there are 60 spurious 1s because the only variable occurrence to propagate to other modules via inter-module propagation is temp in line 35 (variable occurrence number 53), annotated source code is listed in Figure 6.3 On examining matrix $D'$ for this module in Figure 6.2 we can see that none of the spurious 1s to be removed are in column 53, thus it would not be affected.

6.1.4 Quadratic.c

Quadratic.c is a program containing 23 LOC and 3 modules which was translated to C from Fortran by David Wigg. It was originally presented as an example program in Yau and Collofello's 1980 paper [YC80] and used to demonstrate calculation of ripple effect. The program calculates the roots of a quadratic equation given the coefficients $a$, $b$ and $c$. All modules have exactly the same ripple effect for the approximated version as for the Yau and Collofello version. Module main contains only sequential code, modules RROOTS and ROOTS contain sequential and selective code. As in the previous examples no variables are used before being defined, there is therefore no backward referencing and thus no spurious 1s in matrix $D'$.

Ripple effect measures calculated and presented in [YC80] and those produced by REST are shown in Figure 6.4. After finding that the measures produced by REST were not the
### Yau and Collofello calculation:  
- module main = 4  
- module Rroots = 2.9  
- module Iroots = 2.7  
- Program Quadratic(Fortran) = 3.2  

### REST calculation:  
- module main = 2  
- module Rroots = 2.9  
- module Iroots = 2.6  
- Program Quadratic(C) = 2.5

Figure 6.4: Ripple Effect measures for the program Quadratic

same as the measures presented by Yau and Collofello the ripple effect was calculated by hand using the original Yau and Collofello algorithm. The results were identical to those produced by REST. Yau and Collofello were contacted by email to see if they had any memory or evidence remaining of their calculation but unfortunately they have not.

#### 6.1.5 Genscrip.c

Genscrip.c is a string processing program containing 41 LOC and 3 modules. It was written by Frank Langbein as part of the Linux operating system software. The program performs mainly string processing, reading filenames character by character and reproducing them to the screen. All modules have exactly the same ripple effect for the approximated version as for the Yau and Collofello version. No changes are made to matrix D in modules 2 and 3, there are therefore no spurious 1s. In module 1 which contains 2 large loops there are 7 spurious 1s from a total of 29. The reason that the removal of these 1s from matrix D does not affect the ripple effect measured is because all flow in module 1 is through the interface variable ‘word’. Only those variable occurrences which propagate to ‘word’ either directly or via other variable occurrences are picked up. Therefore column 2 in matrix Z which represents all variable occurrences propagating to word in line 2 contains the only relevant entries for computing RE and none are affected by the spurious 1s.
6.1.6 Tamilize.c

Tamilize.c is a string processing program containing 363 LOC and 14 modules. It is a Tamil to TeX converter which was written in 1990 for the University of Washington Humanities and Arts Computing Center, USA. As would be expected in a translation program its tasks are mainly character and string processing. All modules have exactly the same ripple effect for the approximated version as for the Yau and Collofello version. There are only 15 (0.03%) spurious 1's in this program. The code in modules 10 and 14 the two main processing modules is almost entirely contained in for and while loops respectively. Modules 2, 3 and 5 have the highest ripple effect. Module 2 updates the global variable errorcount which is used again in module 14 main, this explains its high ripple effect. Ripple effect for module 3 is high because it is a file handling module which uses pointers and is called in module 14.

6.2 Programs with Differing Ripple Effect

For ten of the fifteen programs in this study Yau and Collofello's ripple effect is different to the approximated ripple effect. This section examines six of these programs, the other four are discussed in Chapter 7. A summary is given of the reasons for the difference and continues with a look at the individual programs in more detail. We look in particular at modules where the Yau and Collofello ripple effect is different from the approximated ripple effect to see if there are particular types of programs where the approximated ripple effect is more or less accurate.
<table>
<thead>
<tr>
<th>Program</th>
<th>Y&amp;C Control</th>
<th>Original</th>
<th>Loops</th>
<th>Branches</th>
<th>No. Mods</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>gensym</td>
<td>16.9</td>
<td>1.9</td>
<td>17.6</td>
<td>5.2</td>
<td>14.3</td>
<td>12</td>
</tr>
<tr>
<td>error</td>
<td>2.6</td>
<td>0.7</td>
<td>3.4</td>
<td>1.3</td>
<td>2.8</td>
<td>12</td>
</tr>
<tr>
<td>bits</td>
<td>5.7</td>
<td>1.0</td>
<td>5.8</td>
<td>2.0</td>
<td>4.8</td>
<td>20</td>
</tr>
<tr>
<td>arippletest</td>
<td>3.0</td>
<td>3.2</td>
<td>3.2</td>
<td>3.2</td>
<td>3.2</td>
<td>5</td>
</tr>
<tr>
<td>fucomp</td>
<td>12.3</td>
<td>1.7</td>
<td>12.4</td>
<td>5.2</td>
<td>8.9</td>
<td>10</td>
</tr>
<tr>
<td>metrics</td>
<td>12.4</td>
<td>0.9</td>
<td>12.9</td>
<td>1.9</td>
<td>12.0</td>
<td>25</td>
</tr>
</tbody>
</table>

Table 6.2: Correlation for programs with differing ripple effect

6.2.1 Reasons for Differing Ripple Effect

From the investigation of the six programs it is apparent that the main cause of difference between Yau and Collofello's ripple effect and the approximated ripple effect is that all definitions of variable occurrences within a module are linked to all uses of variable occurrences, i.e. there is backward propagation. As most modules within programs contain sequential code which is not contained within a selection or iteration and variable occurrences are often used before being defined there are likely to be many modules with spurious 1s. In the programs that we have looked at so far though, this does not seem to cause too much of a difference between the Yau and Collofello ripple effect and the approximated ripple effect (using the Original variant of McCabe). Another factor causing this difference is the number of spurious 1s in matrix D' when variable occurrences are defined and then used in mutually exclusive parts of the modules e.g. when a variable occurrence is defined in the if part of a selection and then used in the else part of a selection. This can also occur within the different cases of a switch statement.

6.2.2 Gensym.c

Gensym.c is a string processing, type conversion and record generation program containing 237 LOC and 3 modules. It was written by Ray Lamb in 1985 as part of the TXEE telephone switching system produced by Nortel for British Telecom. The program uses data from files
1 void Bytesgeneration(Decvalue, Byte1, Byte2) 
2 int *Decvalue; 3 int *Byte1,*Byte2; 
4 { 
5     int Quat; 
6     7 Quat = *Decvalue / 256; 
8     9 if(Quat > 0) 
10     { 
11         *Byte2 = Quat; 
12         *Byte1 = *Decvalue & 256; 
13     } 
14     else 
15     { 
16         *Byte2 = 0; 
17         *Byte1 = *Decvalue; 
18     } 
19 }

Figure 6.5: Module 8 from gensym.c

input to carry out string processing and number conversion then generates a symptom table with one record per symptom. Of the twelve modules within the program the ripple effect measure differs in five. Figure 6.5 shows a typical module from Gensym.c. In all but one of the modules with the same ripple effect there is either no output from the module at all or there is a loop containing most code within the module. Approximated ripple effect for modules 7, 8, 9, and 12 is different from the Yau and Collofello ripple effect. All of these modules contain sequence and selection but no iteration. Backward reference during sequence and selection accounts for all spurious 1s in these modules - 350 (51%) overall, 312 of these occur in module 12 which has 128 variable occurrences.

6.2.3 Error.c

Error.c is a program written on 20/06/94 by Andrew Macpherson and Tej Koonar as part of the TXEE telephone switching system produced by Nortel for British Telecom. Its main function is string processing, printing information to screen about the information processed, it contains 12 modules and 173 LOC. Of the twelve modules within the program the ripple
effect differs in one, there is zero ripple effect in five of the modules. Module 7 which is the only module with differing ripple effect is the largest module within the program containing several ‘if’ ‘else’ statements. All spurious Is are due to backward referencing - 43 overall (32%), 42 of these occur in module 7.

6.2.4 Bits.c

Bits.c is part of the Purdue Compiler Construction Tool Set (PCCTS). Its function is to manage creation and output of bit sets used by the parser, it contains 371 LOC in 20 modules. Bits.c version 1.3 was completed on 26/09/95 by Terence Parr of Purdue University and AHPCRC, University of Minnesota, USA. Of the twenty modules within the program the ripple effect measure differs in two. In six of the modules with the same ripple effect, the ripple effect is zero. Modules with differing ripple effect are 8 and 9. Module 8 contains sequential code and one small if statement. 4 spurious Is are removed from a total of 17. Module 9 contains a large for loop encompassing most of the code within the module, there are 14 spurious Is from 117 in total. All spurious Is removed from these two modules are due to backward referencing. The highest number of spurious Is in bits.c occurs in module 11: 34, but as there are no inputs to, or outputs from the module there is no ripple effect.

6.2.5 Arippletest.c

Arippletest.c is a test program containing 34 LOC and 5 modules. It was written by the author to test different aspects of the output from REST. Modules main, add and result have the same ripple effect for the approximated version as for the Yau and Collofello version. Modules subtract and multiply have differing ripple effect. In the two modules that differ variables are used before being defined, for example in Figure 6.6 variable occurrence a in line 8 is redefined in line 9. The McCabe values for all modules in arippletest.c is 1 because
void Multiply(int a,int b)
{
    printf("Multiply two integers\n");
    printf("First number:");
    scanf("%i",\&a);
    printf("Second number:");
    scanf("%i",\&b);
    total=(a*b);
    a=a+b;
    b++;
    add_total=sub_total+total;
    Result(total);
}

Figure 6.6: Module 4 of aripple\test.c

there is no selection or iteration, consequently the values for all versions of the complexity factor (Original, Control, Loops and Branches) are the same. There are very few spurious 1s, and they all refer to backward propagation as there is no selective or iterative code.

6.2.6 Fcomp.c

Fcomp.c is a flowgraph compiler program containing 282 LOC and 10 modules. It was written at the Centre for Systems and Software Engineering in 1990 as part of the QUALMS flowgraph tool. The program carries out several types of checks on a flowgraph which has been passed to it from another part of the tool. Of the ten modules within the program the ripple effect measure differs in only one, module 6 which is the largest module in the program with 141 variable occurrences. Matrix $D'$ for module 6 contains 22 spurious 1s from a total of 179, there are 17% spurious 1s for the program as a whole. All spurious 1s are due to backward referencing. Most modules within fcomp contain a significant amount of selection, a few contain iteration.
int mccabe (root)
new_tree_node *root;
{
  int num_of_nodes, num_of_edges = 0, child, mccabe_value = 0;
  new_tree_node *parent;
  parent = root->children;
  for(child = 0; child < (*root).child_num; child++)
    if (child == 0)
      mccabe_value += mccabe (root->children) -1;
    else
      { maccbe_value += mccabe (parent->siblings) -1;
        parent = parent->siblings;
      }
  num_of_edges = count_flowgraph_edges (*((root).prime));
  mccabe_value += (double) (num_of_edges - num_of_nodes + 2) ;
  return (mccabe_value);
}

Figure 6.7: Module 3 from metrics.c
6.2.7 Metrics.c

Metrics.c is version II.5.2 of a flowgraph compiler program containing 722 LOC and 25 modules. It was written at the Centre for Systems and Software Engineering in 1992 as part of the QUALMS flowgraph tool. The program calculates twenty-four metrics including McCabe (source code for this module is shown in Figure 6.7) for any given flowgraph received as input. Of the twenty-five modules within the program the ripple effect measure differs in six. Most modules contain sequence and selection, some modules contain iteration. Modules with differing ripple effect in metrics.c are not those with the largest amount of spurious 1s. The difference occurs in those modules where there are several variables assigned to/from throughout the program and where particularly where code is selective. Modules 19, 20, 22, 24 and 25 contain either very many, or large, selective statements, where variables defined in the if part of the selection are used in the else part of the selection. This results in many spurious 1s in matrix D' and thus a difference in the approximated ripple effect.

6.3 Spurious 1s

Figure 6.8 shows all programs in decreasing order of spurious 1s as a percentage of the correct amount of 1s for the module. The Pearson correlation coefficient between % spurious 1s and % difference is low at 0.34. The mean percentage of spurious 1s in these eleven programs is 24.27. The program with the highest number of spurious 1s is metrics.c at 73%. Correlation between the Yau and Collofello ripple effect and the approximated ripple effect for metrics.c is 0.996. This seems surprisingly high given the number of spurious 1s within matrix D'. On investigation it is found that 618 of these spurious 1s are contained within one module, metric_main, module 1. They are all due to backward referencing, many variables in this module are repeatedly used therefore there are a lot of definition/use
**Figure 6.8:** Programs in decreasing order of % of spurious 1s

pairings which are not correct. Most of the ripple effect in module 1 is due to intermodule change propagation caused by the global variable `s_structured` which is used several times in module 13 `is_s_structured`. This does not affect the ripple effect for this module because the spurious 1s removed are spurious definition/use pairings for variables which do not propagate outside of the module. None of these affect the ripple effect measure for this module, the Yau and Collofello measure is the same as the approximated measure.

Gensym.c has the second highest amount of spurious 1s 312 of 350 spurious 1s are contained within its largest module `main` which is module 12. Global file pointers `infile` and `outfile` cause most of the ripple for this module. The approximated ripple effect (17.7) is slightly different to the Yau and Collofello ripple effect (16.6) for this module due to some spurious definition/use pairing between these global variable occurrences. Arippletest has the lowest correlation coefficient: 0.89. and 14% spurious 1s which puts it below the mean amount of spurious 1s: 24.2%.
There is low correlation between the amount of spurious 1s contained within a program and the accuracy of the approximated ripple effect for that program. Therefore we can conclude that there is no obvious link between the two. Of course if there are no spurious 1s then the approximated ripple effect will be identical to the Yau and Collofello ripple effect.

6.4 McCabe Variants and their correlation

The Pearson correlation coefficient is used to see which of the four complexity factors (Control, Original, Loops or Branches) used as part of the approximated ripple effect measure correlate most highly with the Yau and Collofello ripple effect. Correlation is found to be highest using the Original variant of the McCabe measure (section 4.4). The correlation is surprisingly high (0.9974), see Table 6.9 which gives correlation details for all programs used in this study. Correlation for the Branches variant is also high, but lower for Loops.
The correlation for Control is the lowest which shows that we do need to include a complexity factor in our calculation of ripple effect. It seems that our approximated version using Original McCabe, whilst not yielding exactly the same ripple effect as Yau and Collofello, does produce a valid alternative. The spurious 1s produced by not looking at control flow seem not to have much effect on the final result, thus a counterbalance to our approximation is not necessary.

6.5 Summary

This chapter looked at validating approximated ripple effect as a replacement for Yau and Collofello's original ripple effect measure. We showed that our approximated measure is highly correlated with the original ripple effect and as such can replace it. Eleven programs have been used to correlate the two measures: five of these programs had identical Yau and Collofello and approximated ripple effect, six had differing ripple effect and were examined to see what caused the difference in the two measures. The number of spurious 1s in each program was studied to see if there is a relationship between spurious 1s and the accuracy of the approximation. As programs with high amounts of spurious 1s had very high correlation between Yau and Collofello's and the approximated ripple effect there seems to be no obvious link. Pearson correlation coefficient for all versions of the complexity factor were calculated, correlation was highest at 0.9974 for the Original variant of McCabe's cyclomatic complexity. We conclude that our approximated version using Original McCabe, whilst not yielding exactly the same ripple effect as Yau and Collofello, does produce a valid alternative.
Chapter 7

Case Study: A Mutation Testing Software Tool

This chapter describes a case study involving four versions of a mutation testing software tool written in C over a period of several months. The author of the tool was asked to fill in a form after the completion of each version detailing his predicted/intuitive ripple effect for each module. In this chapter successive versions of the tool are examined in detail and the relationship between the approximated ripple effect and a programmer's intuitive ripple effect are compared. Diagrams are presented with modules split into quadrants according to whether they have high/low approximated/predicted ripple effect. These diagrams along with scatter plots and Pearson's correlation coefficient are used to look at patterns within the data. Results show that the main reason for disparity between approximated and predicted ripple effect is that small modules with few variable definitions and high amounts of intermodule change propagation have a higher approximated ripple effect than large modules with a similar amount of intermodule change propagation.

7.1 Overview

We have seen from the previous chapter that approximated ripple effect gives us an accurate approximation of Yau and Collofello's ripple effect measure and thus a valid measure of the
product i.e. the source code. Having satisfied ourselves on this point we then wanted to put our approximation method to practical use in a traditional metrics validation study. Thus we sought a relationship between our product metric and the kind of process attribute it is supposed to capture. Specifically, we aimed to look at whether there was a correlation between approximated ripple effect and a programmer's intuitive ripple effect, in the framework of Fenton [FP96] this is an internal validation. We use a case study of four versions of a software tool comparing each successive version with its predecessors and approximated and predicted ripple effect measures with each other to gain some insight into the problem.

A mutation testing software tool [BW00] was written in 1998 by John Bainbridge, a researcher working in the Centre for Systems and Software Engineering at South Bank University. The tool was produced using the prototyping software development methodology, the first two versions were written in C and the second two were produced using a C++ compiler but are still fundamentally C programs. After producing each version of the tool John was asked to fill in a form (see Figure 7.1) regarding his opinion of the following statement:

"The adaptation of this module may cause problems in the rest of the program".

and asked to give a score of between 1 and 10 for each module within the program. A discussion of actual module changes for this and other programs is given in Chapter 8. John did not have any detailed knowledge of what ripple effect actually measures and was not involved in any other way with our research. As a reminder, conditions necessary for intermodule change propagation (full details were given in section 3.2) to occur are as follows:

1. The variable is a global variable

2. The variable is an output parameter of module m
Name: John Bainbridge
Date: 6/1/98
Program name: Allas1.c
Program date:
Program version: A.1
Number of modules in program: 1

Please list each module in your program then give it a score regarding the following statement:

"The adaptation of this module may cause problems in the rest of the program."

Score within the range 0 - 10 (0 = no foreseeable problems, 5 = some foreseeable problems which could be overcome with a moderate amount of effort, 10 = serious/many foreseeable problems which may be difficult to overcome.)

<table>
<thead>
<tr>
<th>Module number</th>
<th>Module name</th>
<th>Score</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>main</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>make data</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>ran - mutants</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>delete</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>find first line</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>find second line</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>run through bug</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>add one mutant</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>check bugs</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>check bug 2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>add to store</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>add to again</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>print - bug</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>print results</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>print call - bug</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>print - program</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>to char</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>print - store</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>print .cfs</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>main</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

Figure 7.1: Ripple effect form for Allas1.c
<table>
<thead>
<tr>
<th>Program</th>
<th>Y&amp;C Control</th>
<th>Original</th>
<th>Loops</th>
<th>Branches</th>
<th>Modules</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>alias1</td>
<td>15.2</td>
<td>2.2</td>
<td>17.3</td>
<td>6.0</td>
<td>13.5</td>
<td>20</td>
</tr>
<tr>
<td>alias2</td>
<td>18.3</td>
<td>2.6</td>
<td>19.1</td>
<td>7.4</td>
<td>14.2</td>
<td>27</td>
</tr>
<tr>
<td>alias3</td>
<td>20.3</td>
<td>3.8</td>
<td>21.8</td>
<td>10.8</td>
<td>14.8</td>
<td>44</td>
</tr>
<tr>
<td>alias4</td>
<td>19.3</td>
<td>3.7</td>
<td>21.1</td>
<td>10.5</td>
<td>14.2</td>
<td>42</td>
</tr>
</tbody>
</table>

Table 7.1: Pearson correlation coefficient for all versions of tool

3. The variable is an input parameter to a called module.

Intermodule change propagation needs to occur for there to be any ripple effect from one module to another. A description of each version of the mutation testing tool follows. Details are given of the individual modules within the programs, scatter plots of the two measures for each program are presented as the best way to visualise the distribution [Bai93] of approximated vs. predicted ripple effect. Figures 7.4, 7.8, 7.10 and 7.13 show modules within each program split into quadrants according to the amount of approximated vs. predicted ripple effect, Figure 7.2 is shown here as a template. The borders between low/high approximated ripple effect and low/high predicted ripple effect are taken as half of the nearest integer to the highest ripple effect. For example, if the highest approximated ripple effect for a program was 59.6, the nearest integer is 60 and thus the border is set at 30. Modules new to each program are highlighted and the quadrant that a module moved from if appropriate. Quadrants A and D are the most interesting: those with high predicted/low approximated ripple effect and high approximated/low predicted ripple effect. Grouping modules helps us to look at similarities and differences in modules with disparate predicted and approximated ripple effect. This allows us to form some conclusions about why this difference may have occurred. A comparison between programs is given where appropriate and as a summary. Pearson correlation coefficients between Yau and Collofello's ripple effect and our approximated ripple effect, number of modules and LOC for all programs are
in Table 7.1. Correlation coefficients for all programs used in this thesis are shown in 6.9.

7.2 Allas1.c

Figure 7.1 shows the evaluation form for allas1.c which was filled in by John after the program’s completion on 6/4/98. It shows all twenty modules from allas1.c and John’s scores for the modules ranging from 0 to 7. Figure 7.3 is a scatter plot showing the approximated and predicted ripple effect of all modules within allas1.c. The plot shows positive correlation ($r = 0.51$) between predicted and approximated ripple. A line of best fit has been drawn using the Add Trendline facility in Microsoft Excel 97. It shows the relationship between predicted and actual ripple effect in allas1.c which is $y = 0.1x + 1.7$. There follows a description of individual modules within allas1.c. Modules are split into quadrants as shown in Figure 7.4. Eleven of the twenty modules are in quadrant C (low predicted and approximated ripple effect). There follow four sections, one devoted to each quadrant in allas.c giving an idea of the commonalities of modules in the same quadrant.
Quadrant A

Modules that fall in this quadrant have high predicted/low approximated ripple effect. Of the four modules within this quadrant three have a greater than average number of variables, and three have a higher than average McCabe value. Thus modules in this quadrant are typically large and complex. Looking in detail at the approximated ripple effect measure for module *Main* (1) it can be seen that the value for $V_Z X C_{Main}$ is 443 which is high, but because this must be divided by $|V|$ which for this module is 27, the approximated ripple effect is quite low at 16.41. It is possible that John has given these modules a high score due to their length and complexity, but approximated ripple effect is lower than expected due to the modules having many variable occurrences.

*Main* (1) is the harness module for the program as a whole. It calls several other modules and uses two global variables; the second half of the module consists mainly of print statements. *Run_with_many_bugs* (8) reads two parameters as input and has quite complicated control flow, the McCabe value for this module being 14. Several global variables are used in this module accounting for most of the propagation. Module *print_program* (15) contains a 'switch' statement with 13 cases followed by print statements. Module *not_equiv* (20) has
zero approximated ripple effect, but a high predicted ripple effect (5). The McCabe value for this module is 9. Although there is a lot of intramodule change propagation `not_equiv` contains none of the conditions necessary for intermodule change propagation to occur, thus there is no ripple effect.

### Quadrant B

All three modules in quadrant B have large amounts of intermodule change propagation caused mainly by global variables. The modules that are affected by intermodule change propagation have quite high McCabe measures (modules 8 and 17 have McCabe values of 14 and 17 respectively). This increases the final ripple effect quite dramatically. Modules in this quadrant have a lower number of variable occurrences and lower McCabe than all but one of the modules in quadrant A. Thus they are generally smaller and less complex but exhibit ripple via intermodule change propagation to complex modules.

Typical modules from quadrant B are `make_bugs` and `seed.1.mutant`. `Make_bugs` (3) contains one all encompassing 'for' loop which in turn contains three 'for' loops and an 'if else'
statement. The variable bug_no is returned to the calling module main. There are no inputs to the module but one global structure bug which is used throughout the module. Bug accounts for nearly all of the intermodule change propagation from this module, to eight other modules. Propagation from seed_1_mutant (9) comes from the global structure bug and the global array program which propagate through allas1.c to several other modules. It is a short module containing a 'switch' statement with three cases nested within an 'if' statement.

Quadrant C

Modules in this quadrant have low approximated and predicted ripple effect. They split roughly into two groups: those with zero and those with moderate (5-19) approximated ripple effect. Two modules: print_fs (17) and print_store (19) come under the first category. They have approximated ripple effect of zero and predicted ripple effect of zero. Print.fs is a short function containing two 'for' loops which takes variables f and s as input and outputs their values to the screen. Print_store is similar to print_fs, it has two 'for' loops nested within another 'for' loop, its purpose is to print to screen also.

Modules find_first_line (6) and find_second_line (7) fall into the second category. They both have a high number of variable occurrences and a higher than average McCabe (13), thus they are large and complex, similar to the modules described in quadrant A. But, because they have a high number of variable definitions the approximated ripple effect is low.

Quadrant D

Modules in this quadrant have high approximated and low predicted ripple effect. Module run (2) is a sequence of three 'switch' statements each with three cases. Run uses a pointer variable p and a global variable id. The approximated ripple effect for this module is 38.33, the Yau and Collofello ripple effect 20.93 and the predicted ripple effect 3. This is one
void run_1_mutants()
{
    long *correctaddress;
    long answer;
    int i;
    for (i=1; i<=no_of_bugs; i++)
    {
        correctaddress = program[bug[i].location];
        seed_one_mutant(i, TRUE);
        answer = run(program);
        if (answer == rightanswer)
            bug[i].status = ALIVE;
        else
            bug[i].status = KILLED;
        program[bug[i].location] = correctaddress;
    }
}

Figure 7.5: Source code for module Run_1_mutants

instance where there is quite a difference between the approximated and Yau and Collofello ripple effect. If we were using the latter in this analysis this module would have appeared in quadrant C, thus there would be no disparity. Run_1_mutants (4) shown in figure 7.5 is a short module consisting of an 'if else' statement contained within a 'for' loop. The global structure bug again accounts for much of the propagation. It has a low number of variable occurrences and a low McCabe value. Approximated ripple effect is high because there is a lot of propagation from the module to many other modules which have high McCabe values, but it only contains ten variable definitions. Run_1_mutants is the module with the highest approximated ripple effect in allas1.c

7.3 Allas2.c

The evaluation form for allas2.c was filled in after the program's completion on 7/5/98. Figure 7.6 is a scatter plot showing approximated and predicted ripple effect for all modules within allas2.c. Correlation between approximated and predicted ripple effect for allas2.c is lower than allas1.c at 0.17. This section begins with a discussion of the modules from
Figure 7.6: Scatter plot for allas2.c

allas1.c that have been reused in allas2.c followed by a general look at modules which are
new to allas2.c and a comparison of the old and the new modules. This section concludes
with a description of the individual modules within quadrants A, B, C and D within allas2.c.

7.3.1 Modules Shared by Allas1.c and Allas2.c

Allas2.c contains twenty-seven modules, eighteen of the twenty modules from allas1.c (check­
bugs and checkbugs2 are not transferred) and nine new modules. Module numbers are not
necessarily the same as in allas1.c as the modules have been numbered in order of appear­
ance in the code, and for some modules this has changed. In nine of the transferred modules
the approximated ripple effect is higher than it was in allas1.c and in the other nine modules
it is the same. The reason that the approximated ripple effect may be higher even though
the code is the same is because there are more modules using the same global variables.
John’s predicted ripple effect is higher for six of those modules, the same for three modules
and lower for nine of the modules.
All nine modules where the approximated ripple effect has not changed from allas1.c to allas2.c are identical, but the predicted ripple effect for these identical modules is lower for five modules, the same for two modules and higher for three modules. The following comment from the ripple effect form that was filled in after completion of allas2.c may explain the lower scores:

"I think the scores are generally lower than the previous 2 versions. By this time the functions had stabilised, in my mind at least, and I didn't fear so many problems ahead."

N.B. In reference to the 'previous 2 versions' mentioned above, John had already started his second version of the tool before we started this case study, thus we discounted the first version for the purposes of this investigation.

For those modules in both allas1.c and allas2.c where the approximated ripple effect is higher in allas2.c, five of the nine modules are predicted to have lower ripple effect, one is predicted the same and three are predicted higher. The most significant of these are modules: run.1.mutants(7) and seed.one.mutant. The approximated ripple effect for these two modules has risen from 59.40 to 68.40 and 52.47 to 61.88 respectively whereas the ranking for these two modules has gone from 3 to 0 and 4 to 1 respectively. The source code for these two modules is identical. Perhaps John's confidence in his program and the fact that he has not changed these modules from the previous version make him a worse judge of their ripple effect.

7.3.2 Comparison of New and Old Modules

Looking at the scatter plots of modules split into those taken from allas1.c and those new to version two (Figure 7.7), it can be seen that there is a stronger relationship between the approximated and predicted ripple effects for the old modules. The correlation is higher for
old modules. For modules new to allas2 it is 0.08. There seems to be no relation
between approximated ripple effect and predicted ripple effect for these modules new to
allas2. It is perhaps to be expected that a programmer is likely to understand modules
that they have written.

7.3.3 Modules from allas1 reused in allas2

There are six modules, which are reused. They are all predicted to have high
approximated ripple effect, and none of them have low original complexity. There are six 'for'
loops and three 'if' statements which perhaps explains why they are predicted to have high
ripples. There are two remaining modules, which allas (13) and disable (26). These both modules
used for the first time in allas2.

7.3.4 Darst Quadrant Plots

There are seven modules which are used for the first time in allas2. Two of these modules
were predicted to have high approximated ripple effect, one of them to have low approximated
ripples and one of them to have the same. These are four modules.
old modules at 0.20, for modules new to allas2.c it is 0.08. There seems to be no relation between approximated ripple effect and predicted ripple effect for those modules new to allas2.c. It is perhaps to be expected that a programmer is likely to understand modules that they know well and have used before better than those they have just written.

7.3.3 Modules New to Allas2.c

There are nine modules which are new in allas2.c, four of these have approximated ripple effect of zero and predicted ripple effect of 1. Of the other five modules test.if.equivalent (25) has low approximated and predicted ripple effect, both find.equivalents.first (23) and find.equivalents.second (24) have low to moderate approximated ripple effect and high predicted ripple effect. These two modules are similar to each other. They both contain six 'for' loops and three 'if' statements which perhaps explains why they are predicted to have high ripple effect. There are two remaining modules: check.killed (14) and double.line.mutants (26). These both have low predicted but high approximated ripple effect. In both modules most of the propagation is from the global variable program which propagates to eight other modules.

7.3.4 Description of Modules by Quadrant

Quadrant A

There are six modules in this quadrant two of which: main (1) and run.with_many.bugs (11) were previously in this quadrant. Modules find.first.line (9) and find.second.line (10) have moved from quadrant C as their predicted ripple effect has increased from 3 to 4. Modules find.equivalents.first (23) and find.equivalents.second (24) are new to this version. They are very similar to find.first.line and find.second.line thus it is not surprising that they are in the same quadrant.
Figure 7.8: Alls2 modules in Low/High Ripple Effect Quadrants

Quadrant B

Module run is alone in this quadrant. It has moved from quadrant D in allas1.c. to quadrant B in allas2.c because its predicted ripple effect has risen from 3 to 5. Its code is identical in both versions.

Quadrant C

There are fifteen modules in this quadrant, two of which (print_program (18) and not_equiv (27) previously 15 and 20) have moved from quadrant A due to their predicted ripple effect being reduced in allas2.c. Eight modules in this quadrant were here previously and five modules are new to this version. Four of these new modules have zero approximated ripple effect and predicted ripple effect of 1. The remaining module has approximated ripple effect of 6 and predicted ripple effect of 1. All of these new modules are small, with between 2 and 42 variable occurrences with a low McCabe value of between 2 and 7.
Quadrant D

Two modules in quadrant D have moved here from quadrant B: *make.bugs* (3) (details given in 7.2) and *seed.one.mutant*. *Seed.1.mutant* (12) is a small module consisting of a switch statement with three cases. It has a high approximated ripple effect (61.88) caused by the use of two global variables *program* and *id* which are used by many modules. Any module using either of these variables within allas2.c is likely to have high ripple effect because they are used so widely. *Run.1.mutants* (7) is a small module which was present in quadrant D in allas1.c also. It contains an 'if else' statement which is contained within a large 'for' loop. Modules *run* and *seed.1.mutant* are called and there is no variable output from the module. Global variables *bug* and *program* are used. Ripple effect for this module is 68.04 and predicted ripple effect is 0.

There are two modules which are new to allas2.c in this quadrant. *Check.killed* (14) as a module looks similar to *seed.1.mutant*, it is a small module consisting of a switch statement with three cases. It also contains the global variable *program*. Unlike *seed.1.mutant* it calls two of the other modules within allas2.c *run* and coincidentally *seed.1.mutant* and uses its variables as input to those modules. It also has high approximated ripple effect and low predicted ripple effect. There is actually more ripple from *check.killed* than from *seed.1.mutant*, but because when ripple effect is computed the total ripple is divided by the number of definitions the ripple effect for *seed.1.mutant* is higher. *Double.line.mutants* (26) contains four sequential 'for' statements. The global variable *program* is used, modules *run* and *seed.1.mutant* are called and variable *answer* is output from the module. Thus this module displays all the conditions necessary for intermodule change propagation. As expected its ripple effect is high. *Double.line.mutants* has a predicted ripple effect of 1.

It seems that John has an intuitive idea of what causes ripple effect but has given a low ranking to modules with high propagation that are small in size. As previously stated:
John was becoming more confident with his evolving system, he became more confident in the stability of the code he was writing and thus possibly underestimated the ripple effect of several modules.

### 7.4 Allas3.c

The evaluation form for Allas3.c was filled in after the program's completion on 7/7/98. Points on the scatter plot for allas3.c shown in Figure 7.9 almost form two completely separate groups. The group on the left contains modules with low approximated ripple effect, the group on the right contains modules with high approximated ripple effect and low predicted ripple effect. Correlation between approximated and predicted ripple effect for allas3.c is -0.06, the lowest for any of the versions. Looking at 7.10 we can see that there are many more modules in quadrants A and D than in the previous two versions. Twenty of the forty-four modules are in quadrant C, this is similar to the previous two versions where a large proportion of the modules were in this quadrant. There is quite a change in the code for allas3.c, only four modules from allas2.c are reused from the previous version. Looking back at the scatter plot shown in Figure 7.9 we can see that if the right hand group of
modules (roughly those modules in quadrant D) were to move up the y axis by about 3, we would have a much higher correlation and the scatter plot would look more similar to that of allas1.c. Thus, we could posit that John is underestimating in his predictions of ripple effect for a particular group of modules which have high approximated ripple effect. After investigation of these modules it is shown that they are all small modules with lots of intermodule change propagation. We continue by looking in detail at those modules which are similar or the same as those in the previous versions, and then look at modules quadrant by quadrant.

7.4.1 Modules Similar to those in Allas2.c

Eight modules have the same or very similar names to those in allas2.c, we discuss these modules first. Main (1) in allas2.c was the top level module that called most of the other modules consisting of fifteen variable occurrences. In allas3.c Main simply calls module MakePrograms (44) which in turn calls most modules. As such Main in allas3.c cannot be compared with main in allas3.c. Modules Run (8) and MakeBugs (31) occur in both programs, but again there has been a radical change to the modules, thus they cannot be compared. Module FindFirstLine (20) which occurs in both programs is almost exactly the same, differences are due to variable name changes. Approximated ripple effect for both modules is very similar (24.81 and 24.23) for allas2.c and allas3.c respectively, predicted ripple effect is 4 for both modules.

Module FindSecondLine (21) which occurs in both programs is again almost exactly the same, differences are due to variable name changes. This time approximated ripple effect for both modules is quite different (24.03 and 16.48) for allas2.c and allas3.c respectively, predicted ripple effect is 4 for both modules. The difference in the approximated ripple effect is mainly due to the calling of the module run_with_many_bugs in allas2.c by module FindSecondLine, the McCabe measure for this module in allas2.c is 12 whereas for the
corresponding module in allas3.c the McCabe is 6. The modules called are not identical. 

*PrintProgram* (26) occurs in both programs and performs roughly the same task. There is however a large difference in approximated ripple effect between the two modules (0.5 and 31.7) and a difference in the predicted ripple effect: 3 and 0 for allas2.c and allas3.c respectively. Difference in approximated ripple effect is due to the fact that whereas in allas2.c *PrintProgram* ripples to one other module, in allas3.c it ripples to ten modules. Possible explanation for the difference in predicted ripple effect is the previously mentioned statement by John that he became more confident with successive versions of the tool.

Module *NotEquiv* (28) occurs in both programs, the modules are similar though not exactly the same. Approximated ripple effect is 0 in allas2.c and 1.71 in allas3.c and predicted ripple effect 2 and 4 respectively. Difference in approximated ripple effect is due to intermodule change propagation from *NotEquiv*, there is none in allas3.c, predicted ripple effect mirrors this. Module *PrintBugs* (36) occurs in both programs and is similar though not the same. Approximated ripple effect is 2 in allas2.c and 0 in allas3.c and predicted ripple effect 1 and 0 respectively. Difference in approximated ripple effect is due to intermodule change propagation to one other module in allas2.c, there is none in allas3.c, the difference in predicted ripple effect reflects this.

### 7.4.2 Description of Modules by Quadrant

**Quadrant A**

Quadrant A contains eleven modules. One module *not Equiv* moves back into this quadrant as its predicted ripple effect changes back to 4. In allas1.c it was in quadrant A (predicted 4) and in allas2.c it was in quadrant C (predicted 2). The source code for *not Equiv* is not exactly the same as in allas2.c but is extremely similar. Two modules are reused from allas2.c: *FindFirstLine* and *FindSecondLine*, details of these modules are given in section
### Approximated Ripple Effect

![Approximated Ripple Effect Diagram](image)

**Figure 7.10: Alfas3 modules in Low/High Ripple Effect Quadrants**

7.4.1. The remaining eight modules in this quadrant are new in this version. These modules fall into two main groups: large complex modules and small simple modules. There are two modules in the first group. *ProcessProgram* (2) is the harness module for the program alfas3.c. It has ninety-two variable definitions and a McCabe value of 14. It is therefore understandable that it has a high predicted ripple effect, approximated ripple effect though is not high because although there is a lot of ripple from this module there are many variable definitions. Module *MakePrograms* (44) is very similar in size and complexity to *ProcessProgram*. Reasons why the approximated ripple effect is low are also the same.

The second group contain small, simple modules with a small amount of intermodule change propagation. *InitialiseGlobals* (3) is a small, simple module with twenty-three variable occurrences and a McCabe of 4. It initialises the global variables used in the program. *SetInputOutputVars* (4) has one global variable which causes some propagation, but only to one other module *Run* which has a McCabe of 7. The more modules propagated to, the higher the approximated ripple effect. *IncreaseCount* (38) propagates through the global variable *count* which is used extensively in the final module *MakePrograms* (44). Again as there is only intermodule change propagation to one module the approximated ripple effect...
is low.

These modules have all been given high scores by John. It seems intuitively sensible to expect large modules with lots of intermodule change propagation to have a high ripple effect. The smaller modules have global variables but usually only propagate to one or very few other modules. The disparity in approximated and predicted ripple effect has possibly occurred because propagation to many modules dramatically increases the approximated ripple effect for a module.

Quadrant B

Module *GetPrograms* (30) is alone in this quadrant. It is an average size, not particularly complex module with lots of global variable use.

Quadrant C

There are twenty modules in this quadrant, three of which: *FindEquivFirst* (11), *FindEquivSecond* (13) and *PrintBugs* (36) are reused from *allas2.c*. Most other modules are small and simple apart from *ShowResults* (24) and *StoreResults* (25) which are large and moderately complex, they both have more than eighty variable definitions and a McCabe value of 7. Because these modules have no intermodule change propagation their approximated ripple effect is zero.

Quadrant D

All twelve modules in this quadrant have lots of intermodule change propagation, usually to modules that have a high McCabe value. This explains the high approximated ripple effect. Ten of the modules have few variable definitions, thus any ripple that they do have is exaggerated, the other two have many variable definitions and very large amounts of intermodule change propagation. When compared with modules that John has predicted
```c
void SetStatusField()
   { long *correctaddress;
   long answer;
   int i;
   for (i=1; i<=BugCount; i++)
   {correctaddress = Program[Bug[i].location];
      SeedDelta(i,TRUE);
      answer = Run(Program);
      if (answer == RightAnswer)
         Bug[i].status = ALIVE;
      else
         Bug[i].status = KILLED;
      Program[Bug[i].location] = correctaddress;
   }
} // end SetStatusField()
```

Figure 7.11: Module 9 `SetStatusField` from `allas3.c`

as having a high ripple effect it was found that these tended to be similar in content but had more variable occurrences. For example, for module `SetStatusField` (9) shown in Figure 7.11, which is in this quadrant, `VZXGSetStatt`, `Field` is 579 which is moderately high. Because $|V|$ for this module is 10 the approximated ripple effect measure for the module is 57.9, which is the highest approximated ripple effect in this program. By comparison, `ProcessProgram` (2) which is not in this quadrant contains several global variables and inputs to called modules. Looking in more detail at what makes up the approximated ripple effect measure for this module it can be seen that the value for `VZXGProcessProgram` is 981 which is high, but because this must be divided by $|V|$ which for this module is 45, the approximated ripple effect measure for the module is moderate at 21.8. Thus in two modules which look similar in terms of ripple effect there is quite a difference in the approximated ripple effect. The larger module with lots of intermodule change propagation has a lower ripple effect than the smaller module with lots of intermodule change propagation because of the number of variable definitions. It could be that this explains the disparity between the two measures. Perhaps when we look at a large module we expect it to have more impact whereas using
approximated ripple effect the measure will be higher if the module is smaller.

7.5 Allas4.c

All modules in allas4.c have been reused from allas3.c. The scatter plot for allas4.c (Figure 7.12) looks similar to that for allas3.c in the way that the points are spread across the plot. The most noticeable difference is the general reduction in predicted ripple effect. There are two particular changes that we can highlight: some of the modules in the right hand group described in allas3.c have moved up and many of the modules in the left hand group have moved down. Thus there are fewer modules in quadrants A and D (Figure 7.13) and therefore correlation is higher than allas3.c at 0.13. Although allas4.c is very similar to allas3.c it has the most movement of modules from one quadrant to another. This is entirely due to changes in the predicted ripple effect. For modules previously having a predicted ripple effect of 4, seven have decreased in predicted ripple effect (to between 1 and 3) and two have increased to 6. Many of the modules with predicted ripple effect of 3 or 2 in allas3.c have now been given a predicted ripple effect of 1. Predicted ripple effect for allas4.c is 0.29 lower on average than in allas3.c, this may be due to confidence in the
program's stability as previously mentioned in relation to allas2.c.

All but a few modules in allas4.c are identical to those in allas3.c and nearly all changes are cosmetic e.g. a change of module name from SingleLineM (15) in allas3.c to SingleLineMutantsKilled (15) in allas4.c. There are forty-two modules in allas4.c as opposed to forty-four in allas3.c: Getabl2 (42) and Getabc12 (43) have been removed. This has had some effect on the approximated ripple effect of modules as removing modules may change the intermodule change propagation matrix $X_m$.

Differences in approximated ripple effect between allas3.c and allas4.c are minimal and are caused by two factors. Firstly, the change in the McCabe complexity for module MakePrograms (42) which is reduced in size in allas4.c from ninety-six to fifty-nine variable occurrences. The McCabe value decreases from nine to six. Secondly, approximated ripple effect for those modules that were connected to Getabl2 or Getabc12 by intermodule change propagation are affected because the modules are not included in allas4.c.
<table>
<thead>
<tr>
<th>Program Name</th>
<th>Approx. Mean RE</th>
<th>Mean Predicted RE</th>
<th>Mean Number of Variables</th>
<th>Mean McCabe Value</th>
<th>Correlation Coefficient</th>
</tr>
</thead>
<tbody>
<tr>
<td>alias1</td>
<td>17.3</td>
<td>2.7</td>
<td>38</td>
<td>8</td>
<td>0.51</td>
</tr>
<tr>
<td>alias2</td>
<td>19.1</td>
<td>2.0</td>
<td>34</td>
<td>7</td>
<td>0.17</td>
</tr>
<tr>
<td>alias3</td>
<td>21.8</td>
<td>2.4</td>
<td>37</td>
<td>6</td>
<td>-0.06</td>
</tr>
<tr>
<td>alias4</td>
<td>21.1</td>
<td>2.1</td>
<td>36</td>
<td>6</td>
<td>0.13</td>
</tr>
</tbody>
</table>

Figure 7.14: Descriptive information for all versions of the tool

### 7.6 Case Study Analysis and Summary

Figure 7.14 shows correlation between approximated and predicted ripple effect for all alias programs. Correlation between approximated and predicted ripple effect is by far the highest for alias1.c. In quadrant A of alias1.c modules were in general large and complex, predicted ripple effect was relatively higher than approximated ripple effect due to a large number of variable definitions in these modules. One module appeared in quadrant D because of its high approximated ripple effect, if the Yau and Collofello ripple effect had been used the same module would have appeared in quadrant C, thus the disparity for this module is caused by our approximation.

Allas2.c is larger than alias1.c and there is some increase in the amount of approximated ripple effect due to modules being added. Eighteen modules have been reused from alias1.c, correlation between approximated ripple effect and predicted ripple effect is higher for these modules. Mean predicted ripple effect decreases from 2.7 to 1.96, this may be due to John thinking that the modules had stabilised, but correlation between approximated and predicted ripple effect is lower for alias.c. John seems confident in his assertion of the modules stabilising but his new lower predictions are less well correlated than the old ones.
In *allas3.c* modules fall roughly into two groups of data: modules with low approximated ripple effect and modules with high approximated and low predicted ripple effect. All modules in the second group have lots of intermodule change propagation and all but two modules have few variable definitions. Looking at modules that John has predicted high ripple effect for, there is also a lot of intermodule change propagation but the modules are in general larger, they contain more variable definitions. It seems that John’s intuition about what causes ripple effect is correct but does not take into account the fact that approximated ripple may be higher in smaller modules. The $V$ matrix for smaller modules will contain fewer 1s and thus approximated ripple effect will be higher.

Most modules in *allas4.c* are identical to those in *allas3.c*, there is slightly lower approximated ripple effect for the program as a whole due to there being fewer modules. Correlation is higher than for *allas3.c* probably because the modules are reused and thus John again has more confidence in their stability and in his judgement. Predicted ripple effect is lower in general for *allas4.c* than for *allas3.c*. This is noticeable in the scatter plot which looks very similar to that of *allas3.c* but with many of the modules plotted closer to the X axis denoting a decrease in their predicted ripple effect. There are still two groups that can be seen roughly corresponding to those in *allas3.c*.

In general we can summarise as follows: correlation between approximated and predicted ripple effect is significant for *allas1.c* but not for the other versions of the tool. There was a particular disparity between modules with high approximated and low or moderate predicted ripple effect. John seemed to be able to spot modules with lots of intramodule and intermodule change propagation but when this occurred in a small module he did not realise the scaling effect this had on the approximated ripple effect. Thus small programs with high intermodule propagation had lower predicted ripple effect than large modules with high intermodule change propagation. In fact, the smaller programs are more likely to
have higher approximated ripple effect due to the nature of the approximated ripple effect algorithm. Step 9 of our reformulation of the ripple effect algorithm (section 3.4) calls for \( V_m . Z_m . X_m . C \) to be divided by \(|V_m|\), the number of definitions in \( V_m \). For a large module with a lot of intermodule change propagation \( V_m \) will also be large, whereas for a small module \( V_m \) will be small. In practice we found that the result was that in small modules approximated ripple effect tended to be larger.

As John became more confident in the stability of his programs with time his predictions were less accurate. This could be because when writing code programmers have an idea of how much impact any particular piece of code would have on another whereas after some time they forget this impact and are oblivious to side effects because the code works as it should. Another possible explanation is that when asked to complete a task for the first time we pay good attention as it is new to us. Having to complete the task repetitively we become less interested in completing it accurately. Perhaps John thought harder when completing the form for allas1.c than for the subsequent versions.

As to whether there is a correlation between approximated ripple effect and a programmer's intuitive ripple effect I think that the correlation for allas1.c shows there is to some extent. It seems from this case study that when code is first written a programmer has a clear picture of the ramifications of any particular module on other modules within a program, but as they become detached from the code over time this appreciation may decrease. Of course this is only one case study of one programmer and one tool, generalisations cannot be drawn from this analysis. Further case studies may corroborate or invalidate the results of this study.
Chapter 8

Conclusions and Further Work

In this chapter a summary is provided of the outcomes of our research. We describe how our work can be placed in relation to measurement in software maintenance and ripple effect computation. Our motivation was to provide fast automatic computation of ripple effect, this is discussed in more detail in section 8.1. The development of the approximated algorithm is described in section 8.2, and further work to be carried out is outlined in section 8.3. Details of enhancements needed to ensure that REST is an industrial strength tool are described and the building of REST parsers for other programming languages is discussed. The final section 8.4 contains a few concluding remarks.

8.1 Dissertation in a Wider Context

This dissertation has introduced a software measurement tool REST which we hope will benefit the software maintainer and developer by computing ripple effect automatically. Measurement of ripple effect has been incorporated into several software maintenance models to give maintainers valuable information about the code they are maintaining. Maintenance is difficult because it is not clear where modifications have to be made or what the impact will be on the rest of the source code once those changes are made. The ripple effect can be used to help maintainers with assessing that impact. Along with many other
metrics, ripple effect is not the answer to all maintainer's problems, but used as part of a suite of metrics it can give maintainers useful information to make their task easier. Ripple effect is not only useful during software maintenance. During software development it can be compared for different versions of a program to ascertain whether stability is increasing or decreasing and changes made accordingly. In our case study detailed in Chapter 7 ripple effect peaked on the third version and then stabilised in the final version.

Our motivation in this research was to provide a tool to compute ripple effect quickly and completely automatically. Several ripple effect tools are already in existence some of which compute ripple effect without taking intramodule change propagation into account. Others are only semi-automatic, user intervention being required at some point in the computation. Our automatic computation of ripple effect has been at some cost, since we have not taken control flow into account. Therefore we have used several different complexity factors to try and counteract any effect this may have on the final ripple effect measure. The complexity factor which produced the highest correlation with the Yau and Collofello measure in our study was the Original complexity factor. This is in fact the complexity factor used in the original algorithm.

8.2 Development of the Approximation Algorithm

Previous tools developed to produce ripple effect measures have used Yau and Collofello's algorithm which is based on set theory. The intersection and union of sets of variable occurrences is used to calculate paths through source code and measures are produced. It has proved difficult to write simple software using this algorithm, ripple effect tools have either taken an excessive amount of time to produce ripple effect measures or needed some user intervention to make critical decisions about the source code. Our reformulation of the ripple effect using matrix algebra has simplified the algorithm, in effect providing a formal
specification of what was originally a somewhat obscure natural language description. The reformulated algorithm also clarifies the process of computing ripple effect. Each matrix used within the algorithm holds a particular type of information about the software under scrutiny. This makes it easier to understand what each part of the algorithm means and how the ripple effect is being computed.

8.3 Further Work

For REST to be used in industry as a fully working measurement tool some further work needs to be carried out. Facilitation of ripple effect computation for other programming languages besides C is already either at the planning stage or underway. This is detailed in sections 8.3.1 and 8.3.2. We also need to address the robustness of REST. Several enhancements need to be made which are mainly concerned with size and parsing of target source code before REST can be used in industry. The Ripple part of REST that manipulates the matrices to produce the measure needs to be more robust, at the moment it can only cope with modules up to a maximum size of 160 variable occurrences. This highlights the issue of granularity. There are maintenance projects involving programs with modules much larger than 160 variable occurrences which REST should be able to produce measures for.

Further work will also include investigation into the feasibility of measuring ripple effect at different levels of abstraction. So far we have only looked at computing ripple effect using source code at module and program level. Ripple effect computed at subsystem and system level could provide valuable information for use by both developers and maintainers.

An enhancement is needed to the parser which so far has caused the most problems on this project. The parser currently parses array structures as variables, is thus not fine grained enough to parse below the top level of any array to its individual components.
The ripple effect measure where arrays are concerned can be improved. The parser also needs more work to enable all C programs of any version to parse. This is currently being addressed by David Wigg at the Centre for Systems and Software Engineering.

As mentioned in Chapter 3, REST is one of the few tools which specifically gives a ripple effect measure of a program. As it has proved difficult in the past to produce ripple effect automatically, no comprehensive studies have as yet been carried out using automated ripple effect tools to discover how much use ripple effect is to maintainers. Once REST is able to cope robustly with industrial code we aim to carry out such a study.

8.3.1 Ripple Effect of Pascal Source Code

Research was carried out, detailed in [BC00], which looked at the feasibility of building a Pascal parser for REST. The reason behind the choice of Pascal was the availability of the TXE4 telephone switching system from Nortel Networks. TXE4 consisting of half a million lines of code provided us with a unique opportunity to measure an industrial system. Our research showed that there were no major obstacles to the production of a Pascal parser. Several issues were identified: nesting of procedures is allowed in Pascal but not in C, and pointers are not as easily recognisable in Pascal as in C. The configuration of the Pascal parser can be altered slightly from that of the C parser to implement these conditions.

8.3.2 Ripple Effect of Object-Oriented Source Code

Initial investigations are described in [BR01] into whether computation of ripple effect for object oriented code is both meaningful and feasible. Program constructs of Java and C++ were studied to determine whether ripple effect can be computed for object oriented code. If so, a decision needed to be made as to which language to build a parser for first. In Java all classes are chosen at run time, this creates problems for REST as it is a static analysis tool. Therefore because ripple effect computation relies on a certain level of
determinacy, Java programs run the risk of yielding misleading or inaccurate results from ripple effect computations. Polymorphism in Java may also cause problems. The run-time features described for Java are not present in C++. This makes ripple effect computations potentially more reliable for C++ than for Java. C++ has therefore been chosen as the appropriate language for development of a parser.

8.4 Concluding Remarks

This dissertation has presented a simplified algorithm for computing ripple effect and its implementation as a software measurement tool - REST. Computation of ripple effect has been included in several software maintenance models to provide maintainers with invaluable information about the code they are maintaining. As typically 70% [Ben90] of software development budgets are spent on maintenance any help is obviously of great importance. Until now automatic computation of ripple effect has proved troublesome. We aim to make REST robust so that it can be used extensively in industry to compute ripple effect for source code in many languages. The foundations have been built with our work described in this dissertation, our task is now to REST and then enhance our tool.
Bibliography


Appendix A

Data for Allas programs
<table>
<thead>
<tr>
<th>Module</th>
<th>Module name</th>
<th>Approx. Ripple</th>
<th>Ranked Ripple</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>main</td>
<td>16.41</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>run</td>
<td>38.33</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>make bugs</td>
<td>37.54</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>run 1 mutants</td>
<td>59.4</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>getline</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>find first line</td>
<td>24.29</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>find second line</td>
<td>24.07</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>run with many bugs</td>
<td>28.24</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>seed one mutant</td>
<td>52.47</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>checkbugs</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>checkbugs2</td>
<td>32</td>
<td>4</td>
</tr>
<tr>
<td>12</td>
<td>print bug</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>print results</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>print all bugs</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>print program</td>
<td>0.5</td>
<td>4</td>
</tr>
<tr>
<td>16</td>
<td>to char</td>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>17</td>
<td>print fs</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td>add to store</td>
<td>9.61</td>
<td>2</td>
</tr>
<tr>
<td>19</td>
<td>print store</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>not equiv</td>
<td>0</td>
<td>5</td>
</tr>
</tbody>
</table>

**Correlation:** 0.51

Figure A.1: Allas1.c data
<table>
<thead>
<tr>
<th>Module</th>
<th>Module name</th>
<th>Approx. Ripple</th>
<th>Ranked Ripple</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>main</td>
<td>29</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>run</td>
<td>45.67</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>make Bugs</td>
<td>39.73</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>is rhs</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>is const</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>not same</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>run 1 mutants</td>
<td>68.4</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>getline</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>find first line</td>
<td>24.81</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>find second line</td>
<td>24.03</td>
<td>4</td>
</tr>
<tr>
<td>11</td>
<td>run with many bugs</td>
<td>33.66</td>
<td>6</td>
</tr>
<tr>
<td>12</td>
<td>seed one mutant</td>
<td>61.88</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>check positions</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>check killed</td>
<td>53.85</td>
<td>2</td>
</tr>
<tr>
<td>15</td>
<td>print bug</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>print results</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>17</td>
<td>print all bugs</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>18</td>
<td>print program</td>
<td>0.5</td>
<td>3</td>
</tr>
<tr>
<td>19</td>
<td>tochar</td>
<td>17</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>print fs</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>21</td>
<td>add_to_store</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>22</td>
<td>print store</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>23</td>
<td>find equivalents first</td>
<td>16.71</td>
<td>4</td>
</tr>
<tr>
<td>24</td>
<td>find equivalents second</td>
<td>10.71</td>
<td>4</td>
</tr>
<tr>
<td>25</td>
<td>test if equivalent</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>26</td>
<td>double line mutants</td>
<td>62.82</td>
<td>1</td>
</tr>
<tr>
<td>27</td>
<td>not equiv</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

Correlation: 0.17

Figure A.2: Allas2.c data
<table>
<thead>
<tr>
<th>Module</th>
<th>Module name</th>
<th>Approx. Ripple</th>
<th>Ranked Ripple</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>main</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>ProcessProgram</td>
<td>21.8</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>InitialiseGlobals</td>
<td>16.33</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>setInputoutputVars</td>
<td>3.23</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>SetUpProgram</td>
<td>49.15</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>FindVar</td>
<td>4.4</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>FindOp</td>
<td>8.25</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>Run</td>
<td>39.75</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>SetStatusField</td>
<td>57.9</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>SeedDelta</td>
<td>45.75</td>
<td>3</td>
</tr>
<tr>
<td>11</td>
<td>FindEquivFirst</td>
<td>16.71</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>PositionsOK</td>
<td>39.43</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>FindEquivSecond</td>
<td>11.71</td>
<td>3</td>
</tr>
<tr>
<td>14</td>
<td>StoreIfEquiv</td>
<td>8.78</td>
<td>3</td>
</tr>
<tr>
<td>15</td>
<td>SingleLineM</td>
<td>14.92</td>
<td>2</td>
</tr>
<tr>
<td>16</td>
<td>Seed1Line&amp;Run</td>
<td>49.92</td>
<td>2</td>
</tr>
<tr>
<td>17</td>
<td>Seed2Lines&amp;Run</td>
<td>50.91</td>
<td>2</td>
</tr>
<tr>
<td>18</td>
<td>addToStore</td>
<td>16.53</td>
<td>1</td>
</tr>
<tr>
<td>19</td>
<td>PrintStore</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>FindFirstLine</td>
<td>24.23</td>
<td>4</td>
</tr>
<tr>
<td>21</td>
<td>FindSecondLine</td>
<td>16.48</td>
<td>4</td>
</tr>
<tr>
<td>22</td>
<td>RunMutant</td>
<td>20.91</td>
<td>3</td>
</tr>
<tr>
<td>23</td>
<td>MutantKilledOK</td>
<td>17.83</td>
<td>4</td>
</tr>
<tr>
<td>24</td>
<td>ShowResults</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>25</td>
<td>StoreResults</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>26</td>
<td>PrintProgram</td>
<td>31.7</td>
<td>0</td>
</tr>
<tr>
<td>27</td>
<td>IntToChar</td>
<td>17</td>
<td>2</td>
</tr>
<tr>
<td>28</td>
<td>NotEquiv</td>
<td>1.71</td>
<td>4</td>
</tr>
<tr>
<td>29</td>
<td>CheckMutantsKilledInLine</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>30</td>
<td>GetPrograms</td>
<td>32</td>
<td>6</td>
</tr>
<tr>
<td>31</td>
<td>MakeBugs</td>
<td>20.67</td>
<td>4</td>
</tr>
<tr>
<td>32</td>
<td>LvarBugs</td>
<td>41.82</td>
<td>1</td>
</tr>
<tr>
<td>33</td>
<td>RvarBugs</td>
<td>40.8</td>
<td>1</td>
</tr>
<tr>
<td>34</td>
<td>OpBugs</td>
<td>41.82</td>
<td>1</td>
</tr>
<tr>
<td>35</td>
<td>ConstantBugs</td>
<td>40.8</td>
<td>1</td>
</tr>
<tr>
<td>36</td>
<td>PrintBugs</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>37</td>
<td>AddVarToList</td>
<td>21.5</td>
<td>4</td>
</tr>
<tr>
<td>38</td>
<td>IncreaseCount</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>39</td>
<td>GetOp</td>
<td>20.4</td>
<td>2</td>
</tr>
<tr>
<td>40</td>
<td>Get12</td>
<td>17.5</td>
<td>2</td>
</tr>
<tr>
<td>41</td>
<td>Geta12</td>
<td>20.4</td>
<td>2</td>
</tr>
<tr>
<td>42</td>
<td>Getab12</td>
<td>22</td>
<td>2</td>
</tr>
<tr>
<td>43</td>
<td>Getabc12</td>
<td>22.86</td>
<td>2</td>
</tr>
<tr>
<td>44</td>
<td>MakePrograms</td>
<td>13.94</td>
<td>4</td>
</tr>
</tbody>
</table>

**Correlation**: -0.06

*Figure A.3: Allas3.c data*
<table>
<thead>
<tr>
<th>Module</th>
<th>Module name</th>
<th>Approx. Ripple</th>
<th>Ranked Ripple</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>main</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>ProcessProgram</td>
<td>20.2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>InitialiseGlobals</td>
<td>16.33</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>SetUpProgram</td>
<td>3.23</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>FindVar</td>
<td>49.15</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>FindOp</td>
<td>4.4</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>Run</td>
<td>8.25</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>SeedDelta</td>
<td>39.75</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>FindEquivFirst</td>
<td>57.9</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>FindEquivSecond</td>
<td>45.75</td>
<td>4</td>
</tr>
<tr>
<td>11</td>
<td>Seed2Lines&amp;Run</td>
<td>16.71</td>
<td>6</td>
</tr>
<tr>
<td>12</td>
<td>PositionsOK</td>
<td>39.43</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>StoreIfEquiv</td>
<td>11.71</td>
<td>6</td>
</tr>
<tr>
<td>14</td>
<td>SingleLineMutantsKilled</td>
<td>14.92</td>
<td>2</td>
</tr>
<tr>
<td>15</td>
<td>Seed1Line&amp;Run</td>
<td>49.92</td>
<td>3</td>
</tr>
<tr>
<td>16</td>
<td>Seed2Lines&amp;Run</td>
<td>50.91</td>
<td>3</td>
</tr>
<tr>
<td>17</td>
<td>AddToStore</td>
<td>16.53</td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td>PrintStore</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td>FindFirstLine</td>
<td>24.23</td>
<td>6</td>
</tr>
<tr>
<td>20</td>
<td>FindSecondLine</td>
<td>16.48</td>
<td>6</td>
</tr>
<tr>
<td>21</td>
<td>RunMutant</td>
<td>19.82</td>
<td>3</td>
</tr>
<tr>
<td>22</td>
<td>MutantKilledOK</td>
<td>17.83</td>
<td>1</td>
</tr>
<tr>
<td>23</td>
<td>ShowResults</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>24</td>
<td>StoreResults</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>25</td>
<td>PrintProgram</td>
<td>31.7</td>
<td>0</td>
</tr>
<tr>
<td>26</td>
<td>IntToChar</td>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>27</td>
<td>NotEquiv</td>
<td>1.71</td>
<td>3</td>
</tr>
<tr>
<td>28</td>
<td>CheckMutantsKilledInLine</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>29</td>
<td>GetPrograms</td>
<td>32</td>
<td>2</td>
</tr>
<tr>
<td>30</td>
<td>MakeBugs</td>
<td>20.67</td>
<td>2</td>
</tr>
<tr>
<td>31</td>
<td>LvarBugs</td>
<td>41.82</td>
<td>1</td>
</tr>
<tr>
<td>32</td>
<td>RvarBugs</td>
<td>40.8</td>
<td>1</td>
</tr>
<tr>
<td>33</td>
<td>OpBugs</td>
<td>41.82</td>
<td>1</td>
</tr>
<tr>
<td>34</td>
<td>ConstantBugs</td>
<td>40.8</td>
<td>1</td>
</tr>
<tr>
<td>35</td>
<td>PrintBugs</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>36</td>
<td>AddVarToList</td>
<td>21.5</td>
<td>2</td>
</tr>
<tr>
<td>37</td>
<td>IncreaseCount</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>38</td>
<td>GetOp</td>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>39</td>
<td>Get12</td>
<td>10.5</td>
<td>1</td>
</tr>
<tr>
<td>40</td>
<td>Geta12</td>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>41</td>
<td>MakePrograms</td>
<td>14.47</td>
<td>2</td>
</tr>
</tbody>
</table>

**Correlation**: 0.13

Figure A.4: Allas4.c data
Appendix B

Yau and Collofello’s original algorithm
**Step 1** - Create the set $V_k$ for each module $k$ by identifying the variable definitions, i.e. those variables satisfying the following criteria:

a) The variable is defined in an assignment statement.
b) The variable is assigned a value which is read as input.
c) The variable is an input parameter to module $k$.
d) The variable is an output parameter from a called module.
e) The variable is a global variable.

Each variable definition is uniquely identified in $V_k$, thus if the same variable is defined twice within a module then $V_k$ contains a unique entry for each definition.

**Step 2** - Create the set $T_k$ for each module $k$ by identifying the interface variables, i.e. those satisfying the following criteria:

a) The variable is a global variable.
b) The variable is an input parameter to a called module.
c) The variable is an output parameter of module $k$.

Each input parameter is uniquely identified in $T_k$, thus if variable $x$ is utilised as an input parameter in two different invocations each occurrence of $x$ is regarded as a unique interface variable.

**Step 3** - For each variable definition $i$ in every module $k$ compute the set $Z_{ki}$ of interface variables in $T_k$ which are affected by a modification to variable definition $i$ of module $k$ by intramodule change propagation. Intramodule change propagation involves the flow of program changes within the module as a consequence of the modification.

**Step 4** - For each interface variable $j$ in every module $k$, compute the set $X_{kj}$ consisting of the modules involved in intermodule change propagation as a consequence of affected by interface variable $j$ of module $k$. Intermodule change propagation involves flow of program changes across module boundaries as a consequence of the modification.

**Step 5** - For each variable definition $i$ in every module $k$ compute the set $W_{ki}$ consisting of the set of modules involved in intermodule change propagation as a consequence of modifying variable definition $i$ of module $k$, defined by:
Step 6 - For each variable definition \( i \) in every module \( k \) compute \( LCM_{ki} \) as follows:

\[
W_{ki} = \bigcup_{j \in Z_{kj}} X_{kj}
\]

\[
LCM_{ki} = \sum_{t \in W_{ki}} C_t
\]

Step 7 - Compute \( P \) which may be thought of as the probability that a particular variable definition \( i \) of module \( k \) will be selected for modification.

\[
P = \frac{1}{|V_k|}
\]

Step 8 - For each module \( k \) compute \( LRE_k \) and \( LS_k \):

\[
LRE_k = \frac{1}{|V_k|} \sum_{t \in V_k} LCM_{ki}
\]

\[
LS_k = \frac{1}{|LRE_k|}
\]

Step 9 - Compute potential logical ripple effect (\( LREP \)) and potential logical stability (\( LSP \)):

\[
LREP = \frac{1}{n} \sum_{k=1}^{n} \frac{1}{LRE_k}
\]

where \( n \) is the number of modules in the program. Then

\[
LSP = \frac{1}{LREP}
\]
Appendix C

Element Comparison Table
<table>
<thead>
<tr>
<th>Y &amp; C</th>
<th>Black</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V_k$</td>
<td>$V_m$</td>
</tr>
<tr>
<td>$T_k + Z_{ki}$</td>
<td>$Z_m$</td>
</tr>
<tr>
<td>$X_{kj} + W_{kj}$</td>
<td>$X_m$</td>
</tr>
</tbody>
</table>