Available online at www.sciencedirect.com

ScienceDirect

Information and Software Technology 50 (2008) 723–736

INFORMATION AND SOFTWARE TECHNOLOGY

www.elsevier.com/locate/infsof

Deriving an approximation algorithm for automatic computation of ripple effect measures

Sue Black *

Department of Information and Software Systems, Harrow School of Computer Science, University of Westminster, Watford Road, Northwick Park, Harrow HA1 3TP, UK

Received 25 July 2006; received in revised form 2 July 2007; accepted 17 July 2007
Available online 31 August 2007

Abstract

The ripple effect measures impact, or how likely it is that a change to a particular module may 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. Central to this paper is a reformulation in terms of matrix arithmetic of the original ripple effect algorithm produced by Yau and Collofello in 1978. The main aim of the reformulation is to clarify the component parts of the algorithm making the calculation more explicit. The reformulated algorithm has been used to implement REST (Ripple Effect and Stability Tool) which produces ripple effect measures for C programs. This paper describes the reformulation of Yau and Collofello's ripple effect algorithm focusing on the computation of matrix Z_m which holds intramodule change propagation information. The reformulation of the ripple effect algorithm is validated using fifteen programs which have been grouped by type. Due to the approximation spurious 1s are contained within matrix Z_m. It is discussed whether this has an impact on the accuracy of the reformulated algorithm. The conclusion of this research is that the approximated algorithm is valid and as such can replace Yau and Collofello's original algorithm.
© 2007 Elsevier B.V. All rights reserved.

Keywords: Software measurement; Ripple effect; Matrix algebra

1. Introduction

Measurement of ripple effect forms part of an area of fundamental importance to software engineering, that of impact analysis, a type of software measurement. Software measurement as a software engineering discipline has been around now for some thirty years [49]. Its purpose is to provide data that can be used either for assessment of the system in terms of complexity, good structure etc. or prediction of, for example, the total cost of a system 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. It can help to show how effective existing practices are and highlight where improvements are needed [30]. A full description of software measurement and its use is given in [20].

Most software undergoes some change during its lifetime; upgrades to software are common as are changes made to amend or adjust the functionality of a piece of software. For example the software used within mobile phones is upgraded over time to make sure that customers' expectations are met and that particular models of mobile phones can maintain or gain competitive advantage. Software change impact analysis [12] estimates what will be affected in software if a change is made. This information can then be used for planning, making and tracing the effects of changes before the changes are implemented. Examples of impact analysis include [12]:

- Using cross referenced listings to see what other parts of a program contain references to a given variable or procedure.

* Tel.: +44 20 7911 5000x4207.
E-mail address: sueblack@gmail.com

- Using program slicing [43] to determine the program subset that can affect the value of a given variable.
- Using traceability relationships to identify software artefacts associated with a change.

Typically seventy percent of software development budgets are spent on software maintenance [4]. Thus, measures or tools that can speed up the rate at which changes can be made, or facilitate better informed decisions on code changes, can make an important contribution. All types of maintenance involve making changes to source code or its documentation; change impact analysis can show what the effect of that change will be on the rest of the program or system. Software maintenance is difficult because it is not always clear where modifications will have to be made to code or what the impact of any type of change to code may have across a whole system. Change impact analysis via the ripple effect measure has been acknowledged as helpful during software maintenance [9] and as such has been included as part of several software maintenance process models. The usefulness of metrics and models in software maintenance and evolution is described in [15].

In this introduction, ripple effect and change impact analysis and their use during software maintenance have been mentioned, a fuller description is given in Section 2 which describes the background to this work, particularly the history of the ripple effect measure to date. Section 3 discusses measuring complexity and shows where ripple effect fits in. A description of the reformulated ripple effect algorithm is given in Section 4 with details of intramodule and intermodule change propagation and the reformulation's component matrices. Differing versions of matrix C which is used to factor in complexity to the measurement of ripple effect are explained in Section 5 and a validation of the approximated algorithm is provided in Section 6. Section 7 is a description of the programs used in this study. Section 8 summarizes the results of this research and conclusions and further work are put forward in Section 9.

2. Background and related work

The ripple effect measures impact, or how likely it is that a change to a particular module may 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 [39]. The first mention of the term *ripple effect* in software engineering is by Haney in 1972 [23]. He uses a technique called 'module connection analysis' to estimate the total number of changes needed to stabilise a system. Myers [34] 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 [40] also used joint probability of connection 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.

2.1. Yau and Collofello's ripple effect

When Yau and Collofello first proposed their ripple effect analysis technique in 1978 [47] they saw it as a complexity measure that could be used during software maintenance (see Fig. 1) to evaluate and compare various modifications to source code. This work was carried further in 1980 to produce a logical stability measure that is defined as [45, p. 547]:

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

In [45] a software maintenance process is identified (see Fig. 1) where accounting for ripple effect is Phase 3. Other software maintenance models that include ripple effect as part of their lifecycle are detailed in [9]. In the 1980s 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.

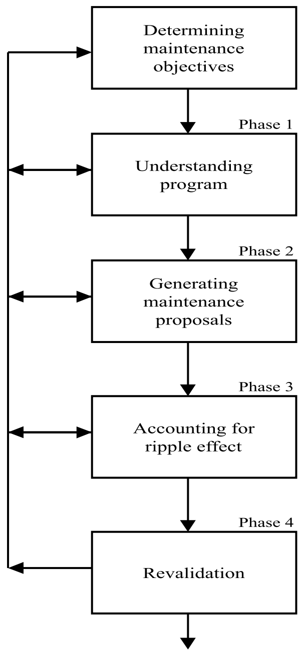
Determining maintenance objectives
Phase 1
Understanding program
Phase 2
Generating maintenance proposals
Phase 3
Accounting for ripple effect
Phase 4
Revalidation

Fig. 1. A methodology for software maintenance [45].