Program Slicing Metrics and Evolvability: an Initial Study

Tracy Hall and Paul Wernick

Systems and Software Group, School of Computer Science
University of Hertfordshire
College Lane, Hatfield, Hertfordshire AL10 9AB, England
tel. ++1707 286323/284782; fax ++1707 284303
{t.hall, p.d.wernick}@herts.ac.uk

Abstract

Previous research has identified a number of metrics derived from program slicing. In this paper we discuss how these metrics relate to the effort required to evolve an existing software-based system. Whilst our interest in this work stems from our development of simulation models of long-term software evolution processes, it will also be directly relevant to the managers of software evolution activities.

Keywords: software evolvability, program slicing, metrics, simulation

1. Introduction

In this paper we investigate how program slicing metric data can be used to measure the evolvability of software systems. We suggest that values for program slicing-based metrics, used in combination with size data, can assist in the prediction of the maintainability of systems over time. This extends our work on modelling and predicting long-term software evolution trends [12].

It is now widely accepted that software systems continue to evolve during their lifetime [6]. The long-term success of such a system depends on its ability to evolve in response to environmental changes. It is also widely accepted that the ability to evolve systems is effectively paramount to their remaining useful [10, p.49].

Historically, measuring the evolvability of software has been performed rather unsatisfactorily. There are currently no generally accepted measures for the evolvability of systems. There has been little input from an underlying theory of software evolution in the derivation of metrics which would allow them to be related to the *evolvability* of software systems.

In our previous work we simulated the long-term evolution of software systems using system dynamics [12]. We found that the difficulty of measuring evolvability with a single metric at any particular time, and changes in that value over time, became a major issue. The lack of metrics which can plausibly reflect the ease or difficulty of evolving an existing software system made that part of our simulation difficult to quantify. As a result, we found it difficult to predict with confidence the impact of a process change on the long-term evolution of a system.

In order to capture the effect of the existing system on further changes to it, we have developed the concept of 'inertia'. We define this as an indirect measure of evolvability which has two dimensions: change (usually growth) in the size of the system as it is evolved, and change to the structure and code of the system as it is evolved. Growth in system size over time may make the system correspondingly more difficult to maintain. However, size alone does not capture the full richness of inertia as a concept, since two systems of equal size may not be equally evolvable.

Meyers and Binkley's work [7] on program slicing-based metrics provides a possible approach to addressing this issue. Meyers and Binkley have conducted longitudinal studies into the behaviour of a number of the metrics described by Weiser [13] and by Ott and Thuss [9]. The use of slicing-based metrics has been proposed previously to focus maintenance interventions and direct re-engineering effort. In this paper we describe an alternative application of slicing data, in which we use these metrics to help quantify the evolvability of software systems rather than as an aid in re-engineering systems. Our focus in this work is towards improving the usefulness of our simulations.

This paper addresses two research questions:

1. Are slice-based measures a viable approach to generating data whose values and trends characterise evolvability?

2. Can evolvability data contribute to the prediction of long-term evolution of software systems?

The rest of the paper is structured as follows. In Section 2 we present an overview of program slicing and describe the main uses of the technique. We provide a summary of existing slice-based metrics in Section 3. We describe in detail the concept of inertia in Section 4, and relate it to our previous models of software evolution processes. We discuss the relationship between slicing metrics and measuring inertia in Section 5. We summarise and conclude in Section 6.

2. Program Slicing

2.1. Program Slicing Introduced

Program slicing was first proposed by Weiser [13, 14] as a technique to assist in debugging programs. The idea emerged in response to Weiser's observations on how experienced debuggers find faults in programs. In its simplest form program slicing identifies all parts of a program that are related to a given statement. This means that all statements that do not affect a particular variable at a specific point in the program are removed. The resulting partial program is referred to as a 'program slice'.

The technique is now supported by a number of code analysis tools. This has encouraged the use of slicing-based techniques in program maintenance, re-engineering, de-bugging and testing. In this paper we argue that, in addition, the technique can also make an important contribution to understanding the evolution of systems in the longer term.

2.2. Program Slicing and Software Evolution

Program slicing is relevant to the evolution of programs because it provides a means of evaluating the implications of changing any line of code in that program. Program slicing captures the *control structure* of a program. It also describes the coupling between lines of code caused by their manipulating shared variable *data*. Program slices thus reflect semantic linkages made in the system by data changes as well as information on the flow of control. Both of these factors are significant in determining how easy or difficult it will be to modify a body of code. The semantic linkages are particularly significant when changes are made which break the encapsulation, information hiding and conformance to fixed interfaces which are characteristic of modern high-quality software designs.

Some program slicing techniques have been developed specifically to improve the quality of software system maintenance work. They allow maintainers to identify, for example, the ripple effects of a program change and thus reduce errors being introduced into the program during maintenance. The underlying rationale of program slicing reflects many important features of software evolvability. Important structural and complexity aspects of a system which are directly relevant to the evolvability of that system are encapsulated in program slices.

Tool support is available to identify slices and enable the localisation of code examination to those parts of the code which need modification and to reflect knock-on, ripple effects [4]. Regression testing, which takes up a considerable proportion of software evolution effort, can also be made easier by slicing-based techniques [5]. Other aspects of software evolution work, including debugging [14] and reverse engineering [1], are also supported using specific slicing-based techniques. Program slicing can also be used to measure directly the cohesion of a program segment [2].

Slicing techniques also make it possible to relate the effort needed to implement an evolutionary change to the structural condition of the system. This suggests that an evolvability measure can be developed by employing slicing-based metrics. This evolvability metric can be used to improve the calibration of our quantitative model of the evolution process [3, 12]. Our motivations for using slicing techniques are, therefore, somewhat different from those of previous researchers. In our work we use slicing as a technique to enhance the understanding of the evolution of a software system. This is in contrast to previous work where slicing is used as a tool to enable maintenance or re-engineering work to be undertaken more effectively, and generally to direct engineering interventions.

3. Slicing Metrics

A number of metrics have been proposed to describe the program slices which can be identified in a system. As noted in Section 2.1 above, slicing-based metrics were first described by Weiser [13] and then extended in the early 1990s by Ott and Thuss [9], in order to characterise the slices which they obtained. Metrics originally proposed by Weiser [13] are described in Table 1. Two further metrics proposed by Ott and Thuss [9] are presented in Table 2.

More recently, tools have become available which allow the collection of larger-scale slicing data. Meyers and Binkley [7] have been the first to collect and analyse such larger-scale data. However the potential for using slicing data in relation to subsequent releases of systems has long been recognised. Ott and Thuss [8] suggested the need for such work.

Table 1. Slicing-based metrics proposed by Weiser [13]

Metric	Description
Coverage	Compares the length of slices to the length of the entire program. Coverage might be expressed as the ratio of mean slice length to program length. A low coverage value, indicating a long program with many short slices, may indicate a program which has several distinct conceptual purposes.
Overlap	Is a measure of how many statements in a slice are found only in that slice. This could be computed as the mean of the ratios of non-unique to unique statements in each slice. A high overlap might indicate very interdependent code.
Clustering	Reveals the degree to which slices are reflected in the original code layout. It could be expressed as the mean of the ratio of statements formerly adjacent to total statements in each slice. A low cluster value indicates slices intertwined like spaghetti, while a high cluster value indicates slices physically reflected in the code by statement grouping.
Parallelism	Is the number of slices which have few statements in common. Parallelism could be computed as the number of slices which have a pair wise overlap less than a certain threshold. A high degree of parallelism would suggest that assigning a processor to execute each slice in parallel could give a significant program speed-up.
Tightness	Measures the number of statements which are in every slice, expressed as a ratio over the total program length. The presence of relatively high tightness might indicate that all the slices in a subroutine really belonged together because they all shared certain activities.

Table 2. Slicing-based metrics proposed by Ott and Thuss [9]

Metric	Description
MaxCoverage	Is the length of the longest slice as a proportion of the program length
MinCoverage	Is the length of the shortest slice as a proportion of the program length

4. Inertia and evolvability

4.1. The concept of inertia

We propose the concept of *Inertia* as a means to characterise the maintainability of a system. It consists of two components, the system size and a measure of the ease or difficulty in changing the system due to its structure and code. Previous work [6, 3] confirms that over the long term systems tend to grow in size, and that as they grow they become correspondingly more difficult to maintain. This is not only because larger systems are likely to be more difficult and costly to maintain than smaller, but also because changes made to software systems over time tend to degrade its structure and makes it less maintainable unless work is performed specifically to counteract this. To model quantitatively how easy a system is to evolve over time, it is important to account for both changes in its size and changes in its structure. Therefore any single quantitative measure of inertia must take account of both of these dimensions.

In our existing simulation models we have used Turski's characterisation of evolutionary growth [11] as the basis for our measure of the effect of changes in the system on the ease of making further changes to it. Turski's calculation, based purely on the physical size of systems, does not directly address the relative evolvability of different systems. In particular, it does not account for issues of system structure and code quality.

In this work we are attempting to capture more of the phenomenon of inertia than Turski's simple abstraction. Program slicing examines, and quantifies in its metrics, the internal linkages of the system which make evolution of one part of a system, without consideration of the rest of it, problematical. Slicing metrics are therefore a good candidate for our purpose.

4.2. System dynamics models of software evolution processes

In our previous work we have used simulation models to help develop an understanding of the long-term evolution of software systems. Figure 1 shows our high-level system dynamics model of a generic software process [12].

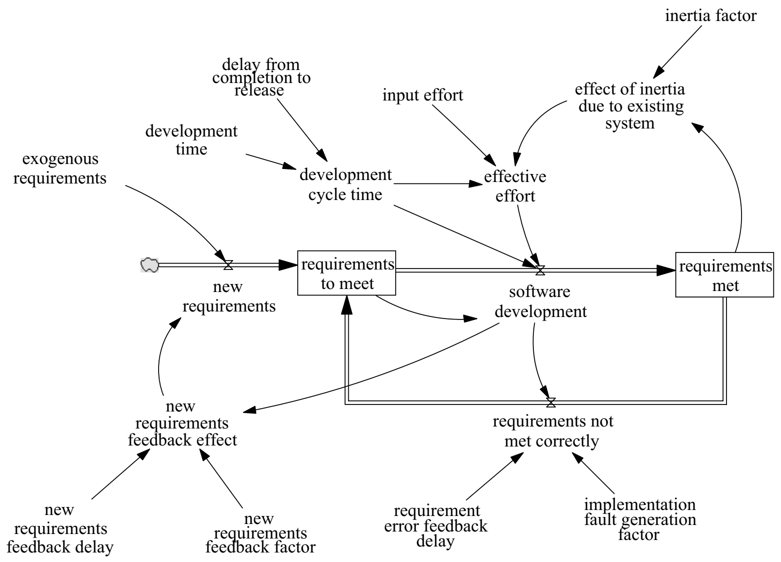
inertia factor
delay from completion to release
input effort
effect of inertia due to existing system
development time
exogenous requirements
development cycle time
effective effort
requirements to meet
requirements met
new requirements
software development
new requirements feedback effect
requirements not met correctly
new requirements feedback delay
new requirements feedback factor
requirement error feedback delay
implementation fault generation factor

Figure 1: the generic software evolution process

In this model the software development process is viewed as a mechanism to convert 'requirements which need to be met' into 'requirements which have been met and fielded to users'. The rate of software development is a function of, *inter alia,* the human resource available to perform evolutionary work and of the inertial of the existing system which slows down that work. This rate of working is subjected to a time delay function to represent the time taken to perform the development work. It is further delayed as completed requirements have to wait until the next release of the software is delivered to its users.

These models have been successful in accurately modelling changes in size of software systems over many years and many releases; see, for example, Chatters *et al.* [3]. Most of the parameters needed to calibrate these models have been obtained from real-world measures of the systems whose behaviour was being investigated, typically either directly from system data or from process experts. The only metric not currently calibrated by these means is the quantified effect of the inertia of the existing system. Using slice-based metrics in addition to the current size-based calculation will allow us to calibrate our models with more precision.

5. Applying slice-based metrics to inertia

In this section we describe how some of Weiser's [13] and Ott and Thuss' [9] slicing-based metrics may be related to the effort needed to evolve a software system. Specifically, we consider the relationship of each metric to the difficulty of making changes to an existing system. In effect, we relate the metric to our notion of the 'inertia' of that system.

- **Coverage**: the existence of many short slices may indicate a system whose structure has been compromised over time by repeated cycles of changes. We conclude that lower coverage implies greater inertia, as more of the code of the system needs to be examined when changing it, i.e. an inverse correlation may be expected between coverage and inertia.

- **Overlap**: higher values of overlap mean that individual elements of code are reused in different traces through the program. Thus, when evolving the system, if a code fragment is identified as needing change, each instance of use of that fragment will need to be located and examined. Even if the required modification does not relate to a specific instance, a change may be needed to the code to support code which still needs the unchanged version. Overall, a direct correlation may be expected between overlap and inertia.
- **Clustering**: lower clustering means higher inertia, because understanding and modifying less well-structured and more mutually interdependent code is likely to be more difficult. This is because the code will be more difficult to understand before changes can be designed. This will lead to greater expenditure of effort and a greater risk of errors being made in the design and implementation of changes. We therefore expect clustering to exhibit an inverse correlation with inertia.
- **Parallelism**: this may indicate that areas of functionality are well-separated in the design and the code. If this is the case, evolutionary changes which respect the existing division of the problem can be made more easily. Therefore, we expect systems exhibiting high parallelism to be more easily evolvable, i.e. the relationship between parallelism and inertia is inverse.
- **Tightness**: this is related to the cohesiveness of the code. As in the case of parallelism, the benefit of more cohesive code can only be exploited if changes which have to be made to a system follow the assumptions implicit in the division of the system functions. In this case, we suggest that it is less likely that a code unit which is truly cohesive will need to be broken up due to the need for system evolution in unexpected directions than is the case for the higher-level design decomposition measured by parallelism. Thus, there may be fewer changes needed overall if the common version can be evolved so as to continue to suit all of its uses. We suggest that code exhibiting high tightness is more likely to be easily evolvable than code with lower tightness.
- **MaxCoverage**: the higher this value, the longer the maximum path length a developer will need to appreciate in order to be able to understand the effect of any change on it and thus evolve the program safely. A high value may also reflect the existence of large blocks of structured code, which is more likely to cause the developer to need to break them up with consequent reworking of code inside a block and the design of new control structures. This metric will therefore be expected to have a direct correlation with inertia.
- **MinCoverage**: a high value for MinCoverage, reflecting a comparatively long 'shortest slice', will be subject to the same problems as those for a high value for MaxCoverage. Conversely, a low value for MinCoverage will mean that at least some evolutionary software changes may be localised to comparatively short traces through the code. We therefore expect MinCoverage also to be directly correlated with inertia.

In quantifying the evolvability of a complete system over time, it may be necessary to select, average, weight and/or total some or all of these measures on the basis of an examination of their trends. At this stage, we consider only the direction (direct/inverse) of the relationship between each metric and inertia, in particular whether there is an inverse or direct relationship between the slice-based data and inertia. Table 3 summarises our findings.

Table 3: The relationship between slicing metrics and inertia

Metric	**Relationship to inertia**
Coverage	Inverse
Overlap	Direct
Clustering	Inverse
Parallelism	Possibly inverse
Tightness	Direct
MaxCoverage	Direct
MinCoverage	Direct

Our conclusions concerning the relationships between these metrics should be seen in the context of Meyers and Binkley's [7] empirical findings. Meyers and Binkley examined, *inter alia*, correlations between slicing metrics obtained for a number of open-source systems. They found strong correlations between

Tightness and MinCoverage and between Tightness and Overlap, and statistically weak correlations between Tightness and Coverage, and MinCoverage and Coverage. They also concluded that Overlap was not correlated to either Coverage or MaxCoverage. They did not consider Clustering and Parallelism. With the exception of our opinion that there is an inverse relationship between Coverage and the other metrics, their results provide some practical support for our arguments.

Their results further suggest that as the size of systems grow, and as they grow older, the deterioration in structure becomes proportionally greater, which lends support to our belief that a relationship exists between trends in slicing metrics and the evolvability of systems, and that slicing metrics can be used as one of the inputs to the calculation of inertia.

6. Conclusions and future work

We have shown that slice-based metrics are a promising way to measure the evolvability of software systems. We have integrated slice-based data with size data to propose inertia as a singe, indirect measure of the evolvability of software systems. We expect this measure of inertia in our system dynamics models to improve the predictions of the long-term evolution of software systems made by these models.

To answer our initial research questions:

1. Are slice-based measures a viable approach to generating data whose values and trends characterise evolvability? Although the work we present here is preliminary, our findings are promising. Slice-based measures look to be a convincing approach to characterising software evolvability. Our re-interpretation of Meyers and Binkley's [7] findings suggests that these metrics will help in quantifying the evolvability of a system.

The work we present here is theoretical, and we will be able to test our answer to this question more fully once we have collected empirical slicing-based metrics data and recalibrated our models. This will extend further the work already done by Meyers and Binkley [7].

2. Can evolvability data contribute to the prediction of long-term evolution of software systems? Again our preliminary results are promising. The addition of evolvability data into our system dynamics models should generate more realistic simulations. This means that our work simulating the long-term evolution of software systems will be capable of being applied with greater confidence to the investigation of the impact of process change on long-term software evolution.

As the next phase of our research, we will collaborate with an industrial partner in generating program slicing metrics to recalibrate and evaluate the model against the evolution of a real-world project.

References

[1] Beck J and Eichmann D, "Program and interface slicing for reverse engineering", *Proc. ICSE 1993*, Baltimore, Maryland, 17–21 May 1993, pp.509–518.

[2] Bieman J and Ott L, "Measuring functional cohesion", *IEEE Trans. Software Eng*., **20** (8), 1994, pp.644–657.

[3] Chatters BW, Lehman MM, Ramil JF, Wernick P, "Modelling A Software Evolution Process", *Software Process: Improvement and Practice*, **5**, 2000, 91–102.

[4] Gallagher KB and Lyle JR, "Using Program Slicing in Software Maintenance", *IEEE Trans Software Engineering*, **17** (8), 1991, pp.751–761.

[5] Gupta R, Harrold MJ and Soffa ML, "An Approach to Regression Testing using Slicing", Proc. *CSM 1992*, Orlando, Florida, 9–12 Nov. 1992, pp.299–308.

[6] Lehman MM, Perry DE, Ramil JF, Turski WM and Wernick PD, "Metrics and Laws of Software Evolution - The Nineties View", *Proc. Metrics '97* Albuquerque, NM, 5–7 Nov, 1997.

[7] Meyers TM and Binkley D, "Slice-Based Cohesion Metrics and Software Intervention", *Proc. IEEE 11th Working Conference on Reverse Engineering,* Delft, Netherlands, 9–12 Nov 2004.

[8] Ott L and Thuss J, "The relationship between slices and module cohesion.", *Proc. ICSE 1989*, Pittsburgh, Pennsylvania, 1989, pp.198–204.

[9] Ott L and Thuss J, "Slice based metrics for estimating cohesion", *Proc. First International Software Metrics Symposium*, Baltimore, MD, May 1993, pp.71–81.

[10] Sommerville I, "Software Engineering", seventh edition, Addison-Wesley, 2004.

[11] Turski WL, "The Reference Model for Smooth Growth of Software Systems Revisited", *IEEE Trans. Software Engineering*, **28** (8), 2002, pp.814 – 815.

[12] Wernick P and Hall T, "The Impact of Using Pair Programming on System Evolution: a Simulation-Based Study", *Proc. ICSM 2004*, Chicago, IL, Sept. 11–14 2004.

[13] Weiser M, "Program slicing", *Proc. ICSE 1981*, San Diego, California, Mar. 9–12 1981, pp.439– 449.

[14] Weiser M, "Programmers use slices when debugging", *Comm. ACM*, **25** (7), 1982, pp.446-452.