JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. 2009; **21**:1–18
Published online 4 November 2008 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/smr.396

Research

An architecture-centric software maintainability assessment using information theory

Muhammad Anan[1,‡], Hossein Saiedian[2,*,†,§] and Jungwoo Ryoo[3,‡]

[1] *Electrical and Computer Engineering, Purdue University Calumet, Hammond, IN 46323, U.S.A.*
[2] *Electrical Engineering and Computer Science, University of Kansas, Lawrence, KS 66045, U.S.A.*
[3] *Information Sciences and Technology, Penn State University, Altoona, PA 16601, U.S.A.*

SUMMARY

Architecture-based metrics can provide valuable information on whether or not one can localize the effects of modification (such as adjusting data flows or control flows) in software and can therefore be used to prevent the changes from adversely affecting other software components. This paper proposes an architecture-centric metric using entropy for assessing structural dependencies among software components. The proposed metric is based on a mathematical model representing the maintainability snapshot of a system. The introduced architectural-level metric includes measures for coupling and cohesion. From this model, the relative maintainability of a component, referred to as a maintainability profile, can be developed to identify architectural decisions that are detrimental to the maintainability of a system. Copyright © 2008 John Wiley & Sons, Ltd.

Received 17 September 2007; Revised 23 June 2008; Accepted 9 August 2008

KEY WORDS: software maintainability; coupling; cohesion; architectural metrics; information theory

*Correspondence to: Hossein Saiedian, University of Kansas, EECS, 2001 Eaton Hall, Lawrence, KS 66045, U.S.A.
†E-mail: saiedian@ku.edu
‡Assistant Professor.
§Professor and Associate Chair.

Copyright © 2008 John Wiley & Sons, Ltd.

1. INTRODUCTION

It is well known that a large fraction of the efforts spent on a software product is devoted to its maintenance [1]. Controlling maintenance costs typically means keeping potential software maintenance hazards from reaching production in an initial release. It also involves monitoring software changes to determine if they introduce maintenance-unfriendly components into an existing system. In order to provide objective and consistent ways to assess software maintainability, appropriate metrics are indispensable.

Boehm *et al.* [2] defined modifiability as the degree to which a system or a component facilitates the incorporation of changes, once the nature of the desired change has been determined.

This paper further explores Boehm's definition by emphasizing that modifiability of a software product is measured by:

- how easy/difficult it is to make changes, and
- how probable it is for an *error* to be introduced when the changes are made.

In many cases, the errors are attributable to architectural decisions made earlier in the development process. For instance, the software could have been developed on a flawed architectural design and is consequently too brittle to allow any serious modifications. It is also possible that the changes made after the initial implementation violate the originally envisioned architecture deliberately designed to accommodate future modifications.

There could be many types of architectural decisions that can influence modifiability, but the authors are particularly interested in the data flow aspect of an architectural decision, which, they believe, is a dominant factor.

Information theory [3] provides a way to measure information entropy (the average number of bits necessary for storing and communicating data). In this paper, the authors demonstrate how entropy can be used to record the reliability of a data flow, which, in turn, becomes a centerpiece of their maintainability metric.

The proposed metric is based on a mathematical model representing the maintainability snapshot of a system. From this model, the relative maintainability of a component, referred to as a maintainability profile, can be developed to identify architectural decisions that are detrimental to the maintainability of a system.

The organization of this paper is as follows. Section 2 provides an overview of the existing methods to measure software maintainability. Section 3 introduces the maintainability metric proposed by the authors. Section 4 presents concrete examples showing how the proposed metric can be applied to real-life software development environments. Section 5 concludes the paper by discussing the limitations of this work and its future extensions.

2. RELATED WORK

Software metrics are used to quantify different aspects of artifacts produced during a software project. A variety of software metrics have been developed to evaluate software architectures [4–14]. Some of these metrics specialize in maintainability and scrutinize maintainability-specific characteristics such as analyzability, changeability, stability, and testability. According to ISO 9126 (an international standard for the evaluation of software):

- *analyzability* represents the effort needed to diagnose deficiencies and identify parts to be modified,
- *changeability* represents the effort needed for modification or fault removal,
- *stability* represents attributes of software that bear on the risk of unexpected effect of modifications, and
- *testability* represents the effort needed to validate modified software.

Software metrics measuring these maintainability characteristics are categorized into either source code assessments of maintainability (analyzability and changeability) or architectural-structure assessments of maintainability (stability and testability) [15–18]. The source-code-based metrics attempt to assess the cognitive complexity of maintaining software, whereas architectural-structure-based metrics attempt to assess the interrelationships and dependencies of software components.

2.1. Source-code-based metrics of maintainability

Source-code-based metrics focus on defining maintainability in terms of the cognitive complexity of the source code. These measures focus on the ability of a programmer to understand and maintain the code itself. The notion of cyclomatic complexity is one of the first measures of such kind of software complexity and hinges on the decision structures of the code as well as the application of algorithms [19].

In addition, building on the idea of cognitive complexity, Halstead [20] developed another source-code-based complexity measure using the number of operators and operands in a software module. In general, this approach focuses more on lexical and textual complexity rather than on structural and logical flows as also shown in McCabe's approach [19].

Welker and Oman [21] proposed a Maintainability Index that is a combination of McCabe's cyclomatic complexity and Halstead's approach. In addition, they worked on identifying specific relationships between software costs and software maintainability.

Berns [22] also concentrated on source code complexity in measuring maintainability and defined his metric as the understandability of the code, which is quantified by the difficulty in grasping how dynamic software parts control their static software counterparts within a software application.

2.2. Architectural-structure-based metrics of maintainability

Architectural-structure-based maintainability metrics provide numerical measures of how facilitating the overall structure (including the interrelationships and dependencies) of software components is in promoting the maintainability of a software application. Such metrics can be developed by specifying a software system in graphs and by deriving measurements from their topologies.

Note that there are few metrics that explicitly claim their use as maintainability metrics per se. Rather, most of them broadly measure complexity (by which maintainability is subsumed) at an architectural level.

Architectural-level metrics of complexity include measures for coupling and cohesion. Coupling is a measure of how strongly one component is connected to or relies on other components. Low coupling implies that modifying a software component has little impact on another component. In contrast to coupling, cohesion measures a component's internal interdependence (i.e., intra-module coupling of the component).

Bieman *et al.* [23,24] proposed a cohesion metric called Design Level Functional Cohesion, which is a high-level review of the inputs and outputs of a component to define its cohesiveness.

Tomlinson [25] proposed a class coupling metric that computes the sum of the number of classes to which a class is coupled divided by the number of its immediate subclasses. Thus, this measure examined the complexity of an inheritance hierarchy in addition to the amount of information contained in associations.

Other researchers also developed metrics for coupling and cohesion in object-oriented systems [26]. Their main contributions lie in finding a relationship between understandability, errors, and probability of errors.

Kazman and Burth [14] argue that coupling and cohesion are not true architectural metrics as they measure the complexity of individual software components with no insight into the overall complexity of a software architecture. They propose a metric based on pattern recognition. More specifically, in their approach, the maintainability of software is judged by the percentage of architectural elements covered by user-defined patterns and the number of distinct patterns utilized in the architecture.

Allen *et al.* [5,6] applied the information theory using coupling and cohesion metrics to evaluate the quality of a design. Their preliminary analysis indicated that the information theory approach makes finer-grain distinctions among alternative graphs than just counting graph features.

In this paper, the objective is to assess structural dependencies among software components in terms of information flow dependency to represent the maintainability snapshot of a system. We sliced the software structures according to their modules' dependencies.

3. THE PROPOSED MAINTAINABILITY METRIC

3.1. Metric construction

In this paper, we propose a metric using the information flow dependencies among software components and slicing the software structures according to these dependencies. The metric serves as a mathematical model of the maintainability state of a system. From this model, the relative maintainability of components (called the maintainability profile) can be constructed to identify architectural decisions detrimental to maintainability.

We use a concrete example (a rail-road crossing system used by Alagar *et al.* [27]) to systematically illustrate how their metric is constructed. The system in the example consists of simple gate controllers. Five trains (`t`) cross through one of two gates (`g`), each of which is controlled by its own controller (`c`). See Figure 1. In the example, each train is connected to only one controller with the exception of only one train (`t3`).

3.1.1. Architectural dependencies

The initial step in the metric construction is the identification of a special type of architectural dependency called *flow* dependency.

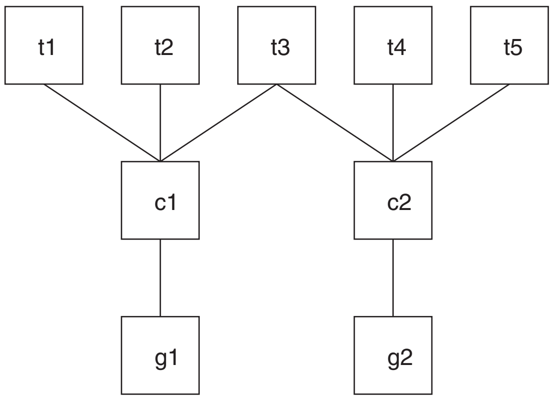


Figure 1. Architecture of a rail-road crossing system.

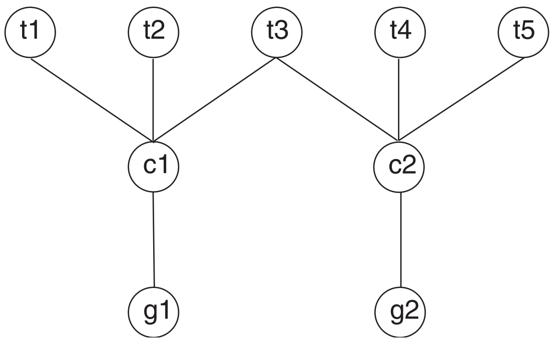


Figure 2. Flow-dependency graph for a rail-road crossing system.

Flow dependencies represent information exchange relationships between components. Any interactions between two components occur by an information flow through a communication channel. Thus, for example, if component u interacts with component v to get data, then there exists an information flow dependency between u and v.

The flow dependencies between components are specified in a graph, in which a component is represented by a vertex. An edge between two vertexes represent a data flow resulting from object references or message exchanges. If there are only two components involved in a directional interaction, a line with an arrow head is used. Otherwise a straight line is drawn.

The end result of the flow dependency analysis is a graph of all software components with every interaction dependency between components displayed. The flow dependencies (representing communication between trains, controllers, and gates) in the rail-road crossing example are shown in Figure 2.

Each train communicates with its respective controller. There is only one train (`t3`) communicating with both controllers. Each controller, in turn, communicates with its respective gate.

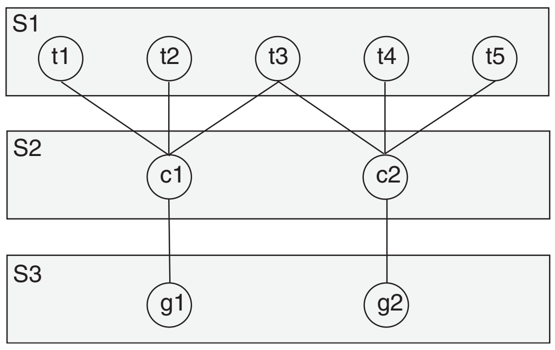


Figure 3. Architectural slicing of the rail-road crossing system graph.

3.1.2. Architectural slicing

The result of the architectural dependency analysis is used to group components. This grouping is referred to as *slicing* and is useful for understanding the design (or the lack of design).

For software maintenance, slicing is significant in that it identifies all the components that can be affected by a software change as well as those that are not affected. In the proposed metric, slicing is used to assess the degree of coupling and eventually the maintainability.

The slicing of the rail-road crossing example (Figure 3) is based on the communications identified by the dependency analysis (Figure 2). The rail-road system is sliced into three subsystems (in a layered fashion) using the similarities in connections between the components:

- sub-module one (S1) contains trains,
- sub-module two (S2) contains controllers, and
- sub-module three (S3) contains gates.

The rail-road crossing system (Figure 3) is a highly simple example of architectural slicing. Note that slicing becomes increasingly difficult and time consuming as the complexity of software grows.

3.1.3. Inter-module and intra-module dependency views

The architectural slicing results are used to construct inter-module dependency (data flows between modules) view and intra-module dependency (data flow within a module) view. Inter-module dependency (for example, components *S1* and *S2*) is shown in Figure 4(a), while intra-module dependency (for example, component *S2*) is shown in Figure 4(b).

The dependency views for the rail-road crossing system (inter-module dependency in Figure 5(a) and intra-module dependency in Figure 5(b)) indicate that there is inter-module dependency, but there is no intra-module dependency. Each layer communicates with its neighboring layer and does not interact with other components in the same layer.

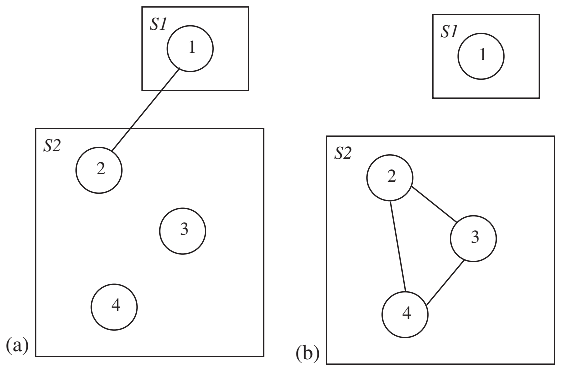


Figure 4. Example of (a) inter-module and (b) intra-module dependency views.

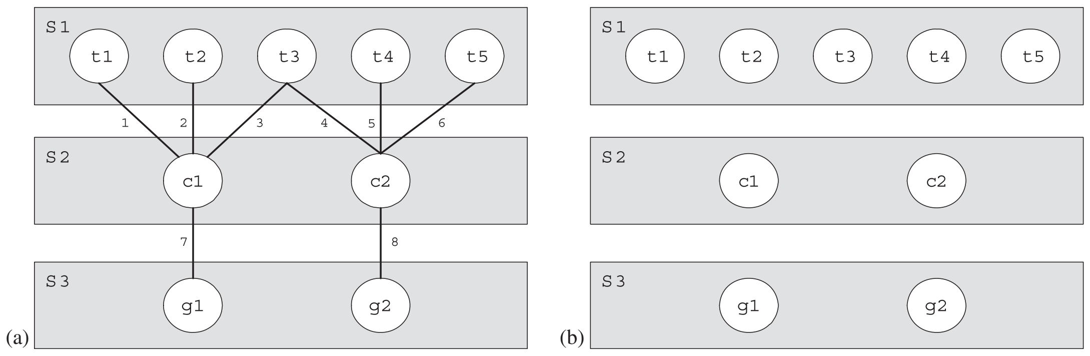


Figure 5. Dependency views of the rail-road crossing system.

3.1.4. Metric definitions

Using the dependency views defined in the previous section, a mathematical model of software maintainability can be established. This model is leveraged to capture the information entropy of software in terms of information flows occurring in the various slicings of a software structure.

Coupling and cohesion are defined by Briand *et al.* [10] in the context of a modular system (MS) graph. Hence, the software-design graph should be partitioned into subsystems (modules). Allen *et al.* [5] define a system graph to explicitly model the lack of relationship between the system and its environment. Given is a set of useful definitions needed to construct a system's mathematical model. These definitions have benefited from [5,10].

Definition 1. MS. An MS is a software system represented by a graph S that has n nodes (components).

Definition 2. Sub-module System. Each MS consists of any number of components denoted as S_i.

Definition 3. Inter-module Dependency View. For a MS (S), the inter-module dependency view is a MS graph whose nodes are connected with edges representing information flows across the boundaries of an architectural slicing.

Definition 4. Intra-module Dependency View. For a MS (S), the intra-module dependency view is a MS graph whose nodes are connected with edges representing information flows within the boundary of an architectural slicing.

Definition 5. Directly Impacted. Let C_k and C_j be two components belonging to either the same or different architectural slicings. C_j is said to be *directly impacted* by C_k if C_j and C_k are connected through a single data flow link.

Definition 6. Indirectly Impacted. Let C_k and C_j be two components belonging to either the same or different architectural slicings. C_j is said to be *indirectly impacted* by C_k if C_j is not directly connected to C_k, but a path $P=[C_1, C_2, \cdots, C_n]$ exists such that C_1 is directly impacted by C_k; C_i is directly impacted by C_{i-1} where $i=2,3,\cdots,n$; C_j is directly impacted by C_n.

Definition 7. Entropy of an Architectural Slicing. Any node i (component) can have R_i number of direct data flow links to other nodes in a system. The information entropy $H(S_i)$ of architectural slicing S_i is defined as

$$\begin{aligned} H(S_i) &= p\sum_{i=1}^{n_s}(-\log(P_L(i))) \\ &= \frac{1}{n}\sum_{i=1}^{n_s}(-\log(P_L(i))) \end{aligned}$$

where k is the number of architectural slicings in a system, $n_{s|S_i}$ the total number of nodes within an architectural slicing S_i, n the total number of nodes in an entire system $n=\sum_{i=1}^{S_k} n_{s|S_i}$, p the probability of a node to be involved in the event of a change $p=1/n$, R_i the total number of inter-module data flow links connected to node i, $P_L(i)$ the probability of a data flow event to occur on the links of a node i and $P_L(i)=1/(R_i+1)$.

Definition 8. System Entropy. For system S with k architectural slicings, its entropy $H(S)$ is defined as

$$H(S)=\sum_{i=1}^{k} H(S_i)$$

where $H(S)$ is the entropy of a system, $H(S_i)$ is the entropy of an architectural slicing S_i and k is the number of architectural slicings in the system.

Definition 9. Information Entropy of an Architectural Slicing.

$$\begin{aligned} I(S_i) &= \sum_{i=1}^{n_s}(-\log P_L(i)) \\ I(S_i) &= \frac{H(S_i)}{p} = n\times H(S_i) \end{aligned}$$

where $I(S_i)$ is the information entropy of an architectural slicing.

Definition 10. Coupling between Architectural Slicings.

$$\text{Coupling}(S_i) = H(S_i)$$

where $H(S_i)$ is the information entropy of architectural slicing S_i.

Definition 11. System Coupling. The cumulative coupling of System S is defined as

$$\text{Coupling}(S) = \sum_{i=1}^{k} H(S_i)$$

where $H(S_i)$ is the information entropy of an architectural slicing S_i, and k is the total number of architectural slicings in a system.

Definition 12. Architectural Slicing Cohesion. The cohesion of architectural slicing S_i is defined as

$$\text{Cohesion}(S_i) = \frac{\text{Coupling}(S_i)}{\text{Coupling}(S)} \quad \text{for } n_s > 1$$

where n_s is the total number of nodes in S_i.

$$\text{Cohesion}(S_i) = 0 \quad \text{when } n_s = 1$$

3.2. Metrics application

Using the metric definitions provided in the previous section, a maintainability profile (value) can be calculated for an entire software system. This maintainability profile combines information entropy values at various abstraction levels into a single, comparable number. The respective maintainability profile values for architectural slicing are simply summed up to produce one maintainability profile value for the system as a whole.

An architectural slicing would have a higher maintainability profile number when it is less desirable for maintenance. A maintainability profile value close to zero indicates that an architectural slice does not require an excessive maintenance effort when a change is introduced.

For the rail-road crossing system, its system coupling (Definition 11) can be calculated by first computing individual coupling values for all the nodes. For example, Coupling($t1$) is calculated as follows:

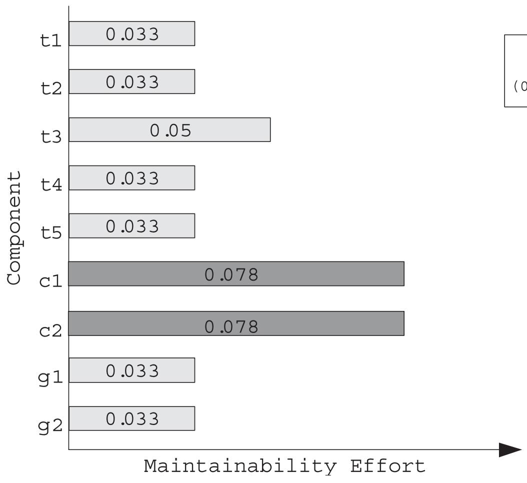
$$H(S_{t1}) = \frac{1}{n} \sum_{i=1}^{n_s} (-\log(P_L)) = \frac{1}{9} \left(-\log\left(\frac{1}{1+1} \right) \right) = \frac{1}{9} (-\log(0.5)) = 0.033$$

This process continues until maintainability profile values for every architectural slicing in the system are computed. From these calculations a maintainability profile value for the whole system emerges (Table I).

Architectural Maintainability Effort (AME) for the system as a whole is then computed: $H(S) = \sum_{i=1}^{3} H(S_i) = 0.404$. The maintainability profile (Table I) can also be visualized for more intuitive reviews (Figure 6). This includes a bar graph (Figure 6(a)) and a *fish-eye* view (Figure 6(b)).

Table I. Maintainability profile for rail-road crossing system.

Slicing S_i	Node n_s	Probability of event occurrence $P_L(i)$	Coupling for a slicing $H(S_i)$	Information entropy $I(S_i)$
S1	$t1$	0.5	0.033	1.69
	$t2$	0.5	0.033	
	$t3$	0.33	0.05	
	$t4$	0.5	0.033	
	$t5$	0.5	0.033	
S2	$c1$	0.2	0.078	1.39
	$c2$	0.2	0.078	
S3	$g1$	0.5	0.033	0.6
	$g2$	0.5	0.033	



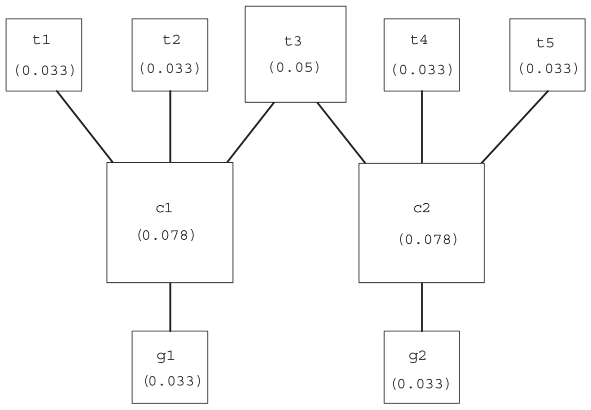


Figure 6. Visual representations of the rail-road crossing system maintainability profile values.

4. A CASE STUDY

Figure 7 shows the result of a flow analysis and the slicing of a software system. Figure 7 is similar to the selected system in [5]. The same system was chosen to show that applying our metric and architectural slicing approach will produce a more accurate estimate of a system's dependencies.

The selection of modules boundary in [5] is made to model both the design decisions to connect each pair of nodes and the decision not to connect the others. The authors generate a set of measures that are sensitive to patterns of connections. In contrast, we used flow dependency to represent information exchange relationships among components. Any interactions between two components occur by an information flow through a communication channel. The end result of the flow dependency analysis is a graph of all software components with every interaction dependency between components displayed. Thus, we construct the architectural slicing in order to group a system's components.

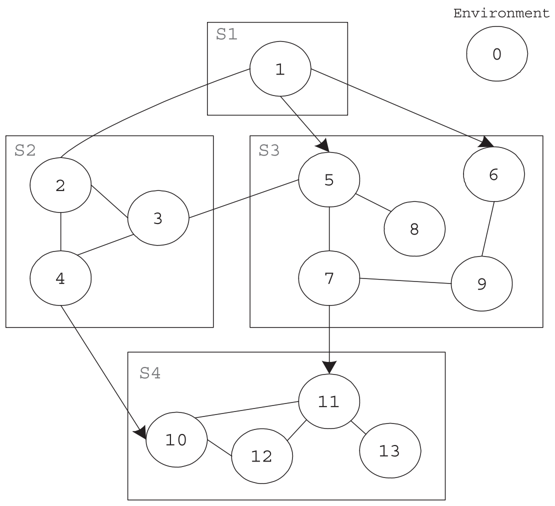


Figure 7. Flow analysis and slicing results of a software system.

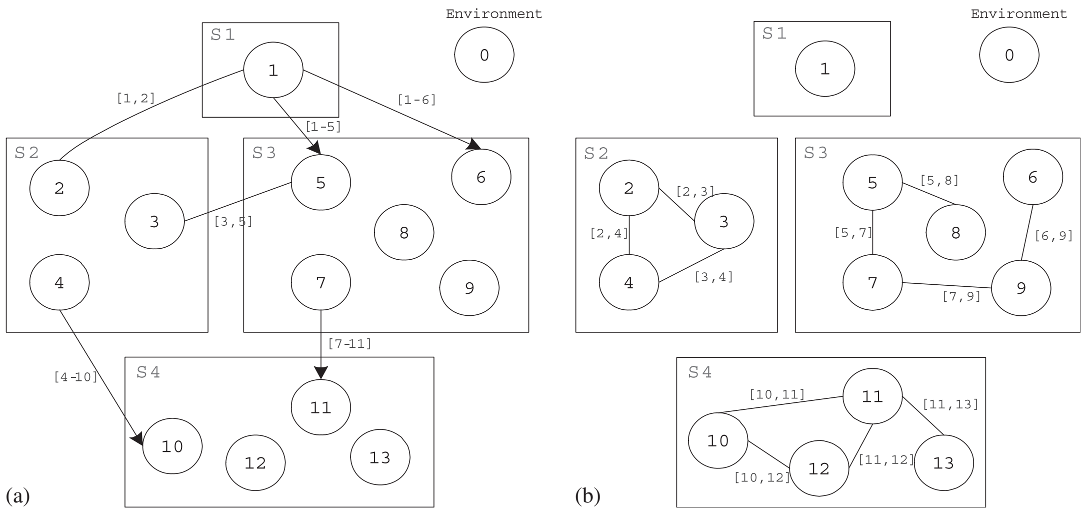


Figure 8. Dependency views.

Figure 8 presents the inter-module and intra-module dependency views of the same system. In this section, more examples with architectural variations are presented to:

- further illustrate some additional concepts including the cohesion of a component, and
- evaluate effects that architectural decisions have on the maintainability profile of a software system.

Table II. Maintainability profile values computed from the inter-module dependency view.

Slicing S_i	Node n_s	Probability of event occurrence	Coupling for a slicing $H(S_i)$	Information entropy $I(S_i)$
Env.	0	1	0	0
S1	1	0.25	0.043	0.6
S2	2	1	0	0.6
	3	0.5	0.021	
	4	0.5	0.021	
S3	5	0.5	0.021	0.6
	6	1	0	
	7	0.5	0.021	
	8	1	0	
	9	1	0	
S4	10	1	0	0
	11	1	0	
	12	1	0	
	13	1	0	

4.1. A software system with strong cohesion

Figure 7 shows the result of a flow analysis and the slicing of a software system. Figure 8 presents the inter-module and intra-module dependency views of the same system. Note that the information links in both dependency views are annotated with labels indicating the directions of information flow. For instance [1,2] means that module 1 sends data to module 2. Table II shows the maintainability profile values of the system computed from the inter-module dependency view (based on Definition 3).

Based on the maintainability profile values (in Table II), AME for the system can be computed as $H(S)=\sum_{i=1}^{3} H(S_i)=0.127$.

The maintainability profile values suggest that architectural slicing *S1* is least maintainable due to its relatively high level of information exchange with other slicings. The most maintainable architectural slicing is *S4* as it exhibits the smallest maintainability profile value (0.00). *S4* does not depend on any other slicing in terms of data flows.

The maintainability profile values from intra-module dependency view (based on Definition 4) can also be computed as shown in Table III.

Using the intra-module maintainability profile values in Table III, one can compute AME for the intra-module dependency view as $H(S)=\sum_{i=0}^{4} H(S_i)=0.378$.

Using Definition 12, one can also compute the cohesion of each architectural slicing (Table IV and Figure 9). The numbers indicate that *S2* is the most cohesive architectural slicing. This makes sense as all the components in the architectural slicing are connected to each other, producing a complete graph.

4.2. Application to various architectures

To illustrate how the proposed metric can be used to evaluate the maintainability of various systems, several system architectures with different component arrangements are constructed (see Figure 10).

Table III. Maintainability profile values computed from the intra-module dependency view.

Slicing S_i	Node n_s	Probability of event occurrence	Coupling for a slicing $H(S_i)$	Information entropy $I(S_i)$
Env.	0	1	0.0	0
S1	1	1	0.0	0.0
	2	0.33	0.034	1.44
S2	3	0.33	0.034	
	4	0.33	0.034	
	5	0.33	0.034	2.05
	6	0.5	0.021	
S3	7	0.33	0.034	
	8	0.5	0.021	
	9	0.33	0.034	
	10	0.33	0.034	1.87
S4	*11*	*0.25*	*0.043*	
	12	*0.33*	*0.034*	
	13	*0.5*	*0.021*	

Table IV. Cohesion values for the architectural slicings.

Slicing S_i	Coupling for a slicing $H(S_i)$	Cohesion for a slicing
S1	0	0
S2	0.102	3.71
S3	0.144	2.63
S4	0.132	2.86

All of these systems have the same number of components (17) and connections (15). However, communication patterns among various modules are different due to the various intra-module dependencies. Therefore, it is not easy to capture the complexity of these architectures without applying a quantitative assessment method to measure the information flow dependency within each system.

Table V summarizes the maintainability profile values for all system architectures shown in Figure 10 using the proposed maintainability metric. The maintainability profile includes: the probability of event occurrence, intermodule coupling, and the total architectural maintainability. The architectures are compared based on the same criteria where sub-modules in each architecture have the same number of communication channels with different components. From the results obtained in Table V, it is clear that sub-module *S3* has the highest complexity value compared with other sub-modules in other architectures and therefore it is the least maintainable. This is because each single component in *S3* refers to other components outside the sub-module in terms of multiple connections. As a result, coupling for *S3* increases. Other sub-modules have different complexities based on their interdependence on other sub-modules.

As illustrated in Figure 10, it is clear that low coupling is always desirable where a component is not dependent on too many other components. Low coupling of a component is a sign of its

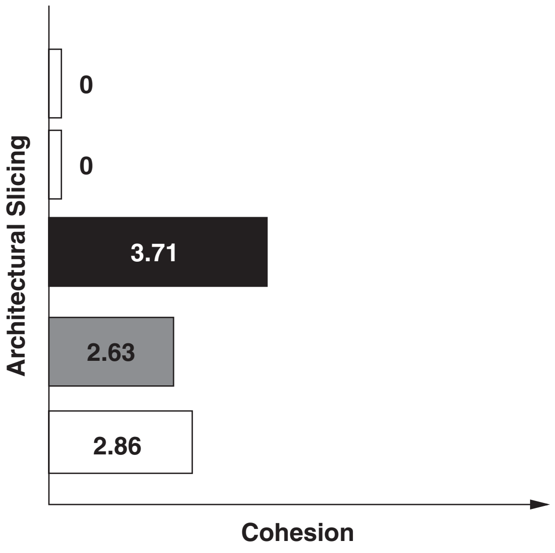


Figure 9. A visual representation of the cohesion values.

independence, which reduces the impact of change. On the other hand, higher value of maintainability (coupling) for a component indicates more interactions between the component and other components of the system, which means harder maintainability for the component. Besides, when the coupling is near zero for a component, the component is easy to maintain.

5. FURTHER RESEARCH AND CONCLUSIONS

The proposed maintainability metric focuses on direct data flow links between components (i.e., those directly impacted) and does not consider the ripple effects of changes among components separated by more than one data flow link (i.e., those indirectly impacted). It would be ideal to take into account both directly and indirectly impacted components, and the authors plan to work on the second incarnation of the metric, which uses a tracing algorithm that runs recursively to identify all the components affected by changes in a particular component [28].

To test the scalability of the metric, the authors also plan to apply their metric to large-scale, industrial-strength software systems implementing various architectural styles. Note that this further validation is not limited to a software system in production but can be done during the different phases of a software development life-cycle, which include design, implementation, and test. One desirable way of doing this type of validation would be to use a software architecture design of known maintainability quality as a benchmark to guide the entire validation process.

To effectively control software maintenance costs, metrics assessing the maintainability state of software systems is crucial. A solid architecture-centric maintainability metric can significantly improve one's ability to accurately evaluate the impact of modifications in software.

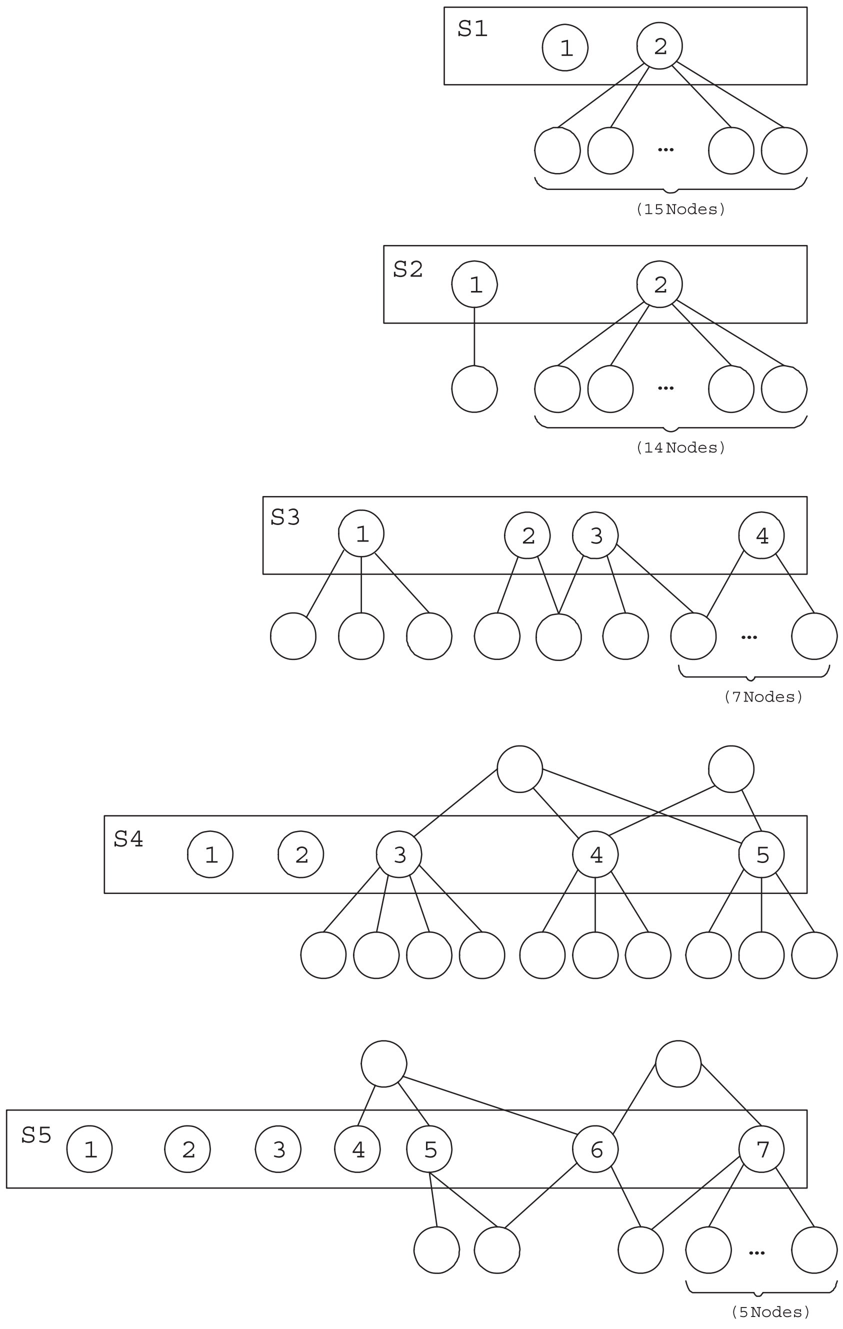


Figure 10. Examples of different module architectures.

Table V. Maintainability profile values for different architectures.

Slicing S_i	Node n_s	Probability of event occurrence	Coupling for a slicing	AME
S1	1	1	0	0.071
	2	0.625	0.071	
S2	1	0.5	0.018	0.087
	2	0.067	0.069	
S3	1	0.25	0.035	0.151
	2	0.33	0.028	
	3	0.25	0.035	
	4	0.125	0.053	
S4	1	1	0	0.138
	2	1	0	
	3	0.17	0.046	
	4	0.17	0.046	
	5	0.17	0.046	
S5	1	1	0	0.147
	2	1	0	
	3	1	0	
	4	0.5	0.018	
	5	0.25	0.035	
	6	0.2	0.041	
	7	0.125	0.053	

This paper proposed a novel maintainability metric based on dependencies (in terms of data flow links) between software components and the information entropies associated with those dependencies. Information entropy is highly relevant to measuring maintainability as increases in information entropy adversely affect the system's responses to modifications. Errors are more easily introduced when the effect of certain changes is difficult to predict, which, in turn, implies poor system maintainability.

REFERENCES

1. Boehm BW, Papaccio PN. Understanding, controlling software costs. *IEEE Transactions on Software Engineering* 1988; **14**(10):1462–1477.
2. Boehm BW, Brown JR, Kaspar H, Lipow M, MacLeod GJ, Merritt MJ. *Characteristics of Software Quality*. North-Holland Publishing: New York NY, U.S.A., 1978.
3. Shannon CE. A mathematical theory of communication. *Bell System Technical Journal* 1948; **27**:379–423, 623–656.
4. Shereshevsky M, Ammari H, Gradetsky N, Mili A, Ammar H. Information theoretic metrics for software architectures. *Proceedings of the 25th International Computer Software and Applications Conference on Invigorating Software Development*. IEEE-CS: Silver Spring MD, October 2001; 151–157.
5. Allen E, Khoshgoftar T, Chen Y. Coupling and cohesion of software modules: An information theory approach. *Proceedings 6th IEEE International Symposium on Software Metrics*, April 2001; 124–134.
6. Allen E, Khoshgoftar T. Measuring coupling and cohesion: An information-theory approach. *Proceedings 7th IEEE International Symposium on Software Metrics*, November 1999; 119–127.
7. Allen EB, Gottipati S, Govindarajan R. Measuring size, complexity, and coupling of hypergraph abstractions of software: An information-theory approach. *Software Quality Journal* 2007; **15**(2):179–212.
8. Abd-El-Hafiz S. Entropies as measures of software information. *Proceedings IEEE International Conference on Software Maintenance*, 2001; 110–117.

9. Allen E, Khoshgoftaar T, Lanning D. An information theory-based approach to quantifying the contribution of a software metric. *Journal of Systems and Software* 1997; **36**:103–113.
10. Briand L, El Emam K, Morasca S. On the application of measurement theory in software engineering. *Empirical Software Engineering* 1996; **1**(1):61–88.
11. Henry S, Kafura K. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering* 1981; **7**(5):510–518.
12. Bianchi A, Caivano D, Lanubile F, Visaggio G. Evaluating software degradation through entropy. *Proceedings of the Seventh International Software Metrics Symposium* (*METRICS'01*), April 2001; 210–219.
13. Carriere SJ, Woods S, Kazman R. Software architectural transformation. *Proceedings of the 6th Working Conference on Reverse Engineering*. IEEE Computer Society: Silver Spring MD, 1999; 13–23.
14. Kazman R, Burth M. Assessing architectural complexity. *Proceedings of the Second Euromicro Working Conference on Software Maintenance and Reengineering* (*CSMR*). IEEE Computer Society Press: Silver Spring MD, March 1998; 104–112.
15. Sheldon F, Jerath K, Chung H. Metrics for maintainability of class inheritance hierarchies. *Journal of Software Maintenance and Evolution*: *Research and Practice* 2002; **14**(3):147–160.
16. Briand L, Lanubile F, Pfleeger S, Rothermel G, Schneidewind N. Empirical studies of software maintenance: A report from WESS '97. *Empirical Software Engineering* 1998; **3**(3):299–307.
17. Meyer B. The role of object-oriented metrics. *IEEE Computer* 1998; **31**(11):123–127.
18. Coleman D, Ash D, Lowther B, Oman P. Using metrics to evaluate software system maintainability. *Computer* 1994; **27**(8):44–49.
19. McCabe TJ. A complexity measure. *IEEE Transactions on Software Engineering* 1976; **2**:308–320.
20. Halstead MH. Elements of Software Science. *Operating*, *and Programming Systems Series*, vol. 7. Elsevier: New York NY, 1977.
21. Welker KD, Oman PW. Software maintainability metrics models in practice. *Crosstalk*, *The Journal of Defense Software Engineering* 1995; **8**(11):19–23.
22. Berns GM. Assessing software maintainability. *Communications of the ACM* 1984; **27**(1):14–23.
23. Bieman JM, Ott LM. Measuring functional cohesion. *IEEE Transactions on Software Engineering* 1994; **20**(8): 644–657.
24. Bieman JM, Kang B-K. Measuring design-level cohesion. *IEEE Transactions on Software Engineering* 1998; **24**(2): 111–124.
25. Tomlinson ZG Jr. Quantifying software maintainability on re-engineered translation of FORTRAN to C++ code. *Master's Thesis*, Florida Institute of Technology, Melbourne, FL, July 2004.
26. Harrison R, Counsell S, Nithi R. Coupling metrics for object-oriented design. *Proceedings of the Fifth International Software Metrics Symposium* (*METRICS'98*). IEEE Computer Society: Silver Spring MD, November 1998; 150–157.
27. Alagar VS, Li Q, Ormandjieva OS. Assessment of maintainability in object-oriented software. *Proceedings of the 39th International Conference and Exhibition on Technology of Object-oriented Languages and Systems*. IEEE Computer Society: Silver Spring MD, 2001; 194–205.
28. Xia F, Srikanth P. A change impact dependency measure for predicting the maintainability of source code. *Proceedings of the 28th Annual International Computer Software and Applications Conference* (*COMPSAC'04*), vol. II, September 2004; 22–23.

AUTHORS' BIOGRAPHIES

Muhammad Anan is an Assistant Professor in the Department of Electrical and Computer Engineering at Purdue University Calumet. Dr Anan has a PhD in Electrical Engineering and Telecommunications Networking from the University of Missouri, Kansas City. He has an MS degree in Electrical and Computer Engineering from the University of Missouri-Columbia and another MS degree in Computer Science from the University of Kansas. Dr Anan has over ten years of industrial experience working for Sprint Nextel and IBM in the fields of telecommunications and Information Technology. Dr Anan's research interests are in the areas of software engineering, computer networks, simulations and modeling, optical network control and switching architectures, network management, digital system design and hardware design languages. He is a member of IEEE.

Hossein Saiedian (PhD, Kansas State University, 1989) is currently a professor of software engineering in the Department of Electrical Engineering and Computer Science at the University of Kansas (KU) and a member of the KU Information and Telecommunication Technology Center (ITTC). Professor Saiedian's primary area of research is software engineering. He has over 130 publications in a variety of topics in software engineering and computer science. Saiedian's research in the past has been supported by the NSF as well as other foundations. He is a Senior member of IEEE.

Jungwoo Ryoo is an Assistant Professor of Information Sciences and Technology at the Pennsylvania State University-Altoona. His research interests include information assurance and security, software engineering, and computer networking. Dr Ryoo conducts extensive research in software security, network/cyber security, security management (particularly in the government sector), software architecture, Architecture Description Languages (ADLs), object-oriented software development, formal methods and requirements engineering. He also has significant industry experience working with Sprint and IBM in architecting and implementing secure, high-performance software for large-scale network management systems. He received his PhD in Computer Science from the University of Kansas in 2005.