On Assessing the Complexity of Software Architectures

Jianjun Zhao

Department of Computer Science and Engineering
Fukuoka Institute of Technology
3-10-1 Wajiro-Higashi, Higashi-ku, Fukuoka 811-0214, Japan
zhao@cs.fit.ac.jp

Abstract

This paper proposes some new architectural metrics which are appropriate for evaluating the architectural attributes of a software system. The main feature of our approach is to measure the complexity of a software architecture by capturing various types of architectural dependences in the architecture that can be derived from analyzing its formal architectural specification.

keywords: Architectural description lanaguage, Architectural metric, Dependence analysis, Software architecture

1 Introduction

Software metrics have many applications in software engineering activities including software analysis, testing, debugging, maintenance, and project management. In the past two decades numerous software metrics have been proposed for measuring the complexity of software [4, 25]. These metrics can be divided into two categories according to the design levels of software[1]: *code metrics* which aim at measuring the complexity of a single program module at code design level [3, 5, 6, 13], and *architectural metrics* which aim at measuring the complexity of components and their interconnections in software systems at architectural design level [7, 10, 14, 22].

Most work on software metrics focused on code metrics which are derived solely from source code of a program, and the study of architectural metrics has received little attention. However, architectural measurement can be regarded as a desirable addition to code metrics because it allows you to capture important aspects of a system's architecture earlier in the system life cycle so you can take corrective actions earlier [16]. This may offer greater potential for return on investment in order to make large gains in productivity and quality since error detection and repair is more costly if we can not catch errors in the early stage of system design.

---

[1] There are usually two levels of design for software, *architectural level design* where involves overall association of system capability with components, and *code level design* where involves algorithms and data structures [16].

But, why has the study of architectural metrics received little attention in comparison with code metrics ? One important reason is while the code level for software systems is now well understood, the architectural level is currently understood mostly at the level of intuition, anecdote, and folklore [19]. Existing representations that a system architect uses to represent the architecture of a software system are usually informal and *ad hoc*, and therefore can not capture enough useful information of the system's architecture. Moreover, with such an informal and *ad hoc* manner, it is difficult to develop analysis tools to automatically support the evaluation and comparison of existing architectural metrics. As a result, in order to make architectural metrics more widely accepted and used in software system design, formal representation of system architectures is strongly needed.

Recently, as the size and complexity of software systems increases, the design and specification of the overall software architecture of a system is receiving increasingly attention. The software architecture of a system defines its high-level structure, exposing its gross organization as a collection of interacting components. A well-defined architecture allows an engineer to reason about system properties at a high level of abstraction [19]. Architecture description languages (ADLs) are formal languages that can be used to represent the architecture of a software system. They focus on the high-level structure of the overall application rather than the implementation details of any specific source module. In order to support formal representation and reasoning of software architecture, a number of ADLs such as WRIGHT [1], Rapide [9], and UniCon [18] have been proposed. By using an ADL, a system architect can formally represent various general attributes of a software system's architecture. This provides researchers with a promising solution to solve the problems existing in recent architectural metrics. First, a sound basis for software architecture promises one to define new architectural metrics, or refine existing architectural metrics in a more formal way in comparing with existing informal structure charts based architectural metrics. Second, formal language support for software architecture provides a useful platform on which automated support tools for architectural metrics can be developed and formal evaluation and comparison of existing architectural metrics can be done.

In this paper, we propose some new architectural metrics for software architecture. Our metrics are appropriate for evaluating the architectural attributes of a software system. The main feature of our approach is to measure the complexity of a software architecture by capturing various types of architectural dependences in the architecture that

can be derived from analyzing its formal architectural specification. To formally define these metrics, we present a dependence-based representation named *Architectural Dependence Graph* (ADG) to explicitly represent various architectural dependences in a software architecture.

The rest of the paper is organized as follows. Section 2 presents three types of architectural dependences in a software architecture and the architectural dependence graph. Section 3 defines some dependence-based metrics for software architecture. Section 4 discusses some related work. Concluding remarks are given in Section 5.

2 A Dependence Model for Software Architecture

When we intend to measure some attributes of an entity, we must build some model for the entity such that the attributes can be explicitly described in the model. In this section, we present a dependence model for software architecture to capture attributes concerning about information flow in a software architecture.

2.1 Program Dependences

Program dependences are dependence relationships holding between program statements (variables) in a program that are implicitly determined by control flow and data flow in the program. Usually, there are two types of program dependences in a program, that is, *control dependences* representing the control conditions on which the execution of a statement or expression depends and *data dependences* representing the flow of data between statements or expressions. The task to determine a program's dependences is called *program dependence analysis.*

Program dependence analysis has been primarily studied in the context of conventional programming languages. In such languages, it is typically performed using a *program dependence graph* [3, 8, 15]. Program dependence analysis, though originally proposed for complier optimization, has also many applications in software engineering activities such as program slicing, understanding, debugging, testing, maintenance and complexity measurement [8, 15, 17]. As a result, it seems reasonable to apply program dependence analysis technique to software architectures to support software architecture development activities [21, 23].

2.2 Architectural Dependences

Roughly speaking, architectural dependences are dependence relationships holding between components (ports) in a software architecture, and are implicitly determined by information flow in the architecture. Unlike program dependences, which are defined as dependence relationships between statements (variables) in a program, architectural dependences are defined as dependence relationships between components (ports) in a software architecture. To perform dependence analysis on software architectures, it is important to identify all primary architectural dependence relationships between components (ports) in the architectures. However, such a work is quite difficult because comparing with program dependence analysis, the dependence relationships between components (ports) in a software architecture can be more complex and broad. In this section we introduce three types of primary architectural dependences between components (ports) in a software architecture. The classification of architectural dependence types based on the

results of *coordination theory* [12] [2]. The types of primary architectural dependences are not limited to these three ones, rather, new types of primary architectural dependences must be further exploited in order to identify all types of primary architectural dependences in a software architecture.

Shared Dependences

Sharing dependences represent dependence relationships among consumers who use the same resource or producers who produce for the same consumers. For example, for two components u and v, if u and v refer to the same global data, then there exists a shared dependence relationship between u and v.

Flow Dependences

Flow dependences represent dependence relationships between producers and consumers of resources. For example, for two components u and v, if u must complete before control flows into v (prerequisite), or if u communicate v by parameters, then there exists a flow dependence relationship between u and v.

Constrained Dependences

Constrained dependences represent constraints on the relative flow of control among a set of activities. For example, for two components u and v, u and v can not execute at the same time (mutual exclusion), then there exists a constrained dependence relationship between u and v.

2.3 Architectural Dependence Graph

We present an arc-classified digraph named *Architectural Dependence Graph* (ADG) for explicitly representing the three types of primary architectural dependences in a software architecture. Here we assume that the interface of each component in a software architecture is defined by a set of *ports*. The ADG of a software architecture consists of vertices and arcs to connect these vertices. There is a *component vertex* for each component in the architecture, and each component vertex consists of a set of *port vertices* each representing a port of the component. There is an architectural dependence arc between two port vertices of components if there exists a shared, flow, or constrained dependence relationship between the ports.

Architectural dependence information can be inferred based on formal architectural specifications of a software architecture. For example, based on a WRIGHT architectural specification we can infer which ports of a component are input ports and which are output ports in the specification. Moreover, the direction in which the information transfers between ports can also be inferred based on the formal specification. Such kinds of information can be used to construct the architectural dependence graph for a software architecture to formally define dependence-based architectural metrics.

3 Architectural Metrics

As we mentioned in Section 2, architectural dependences are dependence relationships holding between components in a software architecture that are implicitly determined by information flow in the architecture. Therefore, architectural dependences can be regarded as one of intrinsic attributes

---

[2] In [12], Malone and Crowston defines *coordination* as the process of *managing dependences* among activities.

of a software architecture and it is reasonable to regard architectural dependences as one of objects for measuring the architectural complexity of a software architecture.

In this section, we define a set of new architectural metrics in terms of architectural dependences to measure the complexity of a software architecture from various different viewpoints. Once the ADG of a software architecture is constructed, the metrics can be computed easily based on the graph. The following notations are used for defining these metrics:

$|A|$: the cardinality of set A.

R^+: the transitive closure of binary relation R.

$\sigma_{[1]=v}(R)$: the selection of binary relation R such that $\sigma_{[1]=v}(R) = \{(v1, v2)|(v1, v2) \in R \text{ and } v1 = v\}$.

When we constructed the ADG for a software architecture, the most general metric can be defined in terms of ADG. The following metric is defined for measuring the total complexity of a software architecture:

- Let D_t be the set of all dependences arcs in the ADG of a software architecture, then the total complexity M_T of the architecture can be measured by $M_T = |D_t|$.

Note that the above metric was defined under the situation that we treat a component as an unit to construct the ADG of a software architecture. However, in fact, each component in the architecture may generally correspond to a single application module which can be measured by usual code metrics at code level. So there is a need to combine the total complexity at architectural design level with internal component complexity at code design level to obtain an overall complexity metric.

- Let M_T be the total complexity and $M_1, ..., M_k$ be the individual component complexities. Then the global complexity M_G of a software architecture can be measured by: $M_G = M_T + \sum_{i=1}^{k} M_i$.

The above metrics only concerned with the direct architectural dependences in a software architecture, but did not take indirect architectural dependences into account. As a result, they only capture the sum of some local complexity, rather than the total complexity of the architecture. In fact, a component in a software architecture may indirectly depend on other components in the architecture. Therefore, to measure the total complexity of a software architecture, we should define a metric by taking either direct or indirect architectural dependences into account. This can be obtained by calculating the transitive closure $|D_t^+|$ of the $|D_t|$, we have: $M_T' = |D_t^+|$. Similarly, if we also consider the indirect dependences at architectural level and each of application modules at code level, we can obtain more detailed global complexity M_G' of the system: $M_G' = M_T' + \sum_{i=1}^{k} M_i'$.

In maintenance phases, when we have to modify some component in a software architecture, usually, we intend to know information about how the modified component intersect with other components. This kind of information is very useful because it can tell us if the modified component is a special point that connects with its environment more closely than other components. If so, that means it is difficult to make changes to the component due to a large number of potential effects on other components. We call such a component the "most easily affected component of the architecture." To capture such attribute, we can define following metrics.

- Let D_t be the set of all dependences arcs in the ADG of a software architecture, and $\sigma_{[1]=v}(D_t)$ be the number of ports of components on which a port v of a component is directly dependent. The complexity M_S of the most easily affected component in the architecture can be measured by $M_S = max\{|\sigma_{[1]=v}(D_t)| \mid v \text{ is a vertex of the ADG }\}$.

Similarly, if we also considered indirect architectural dependences in a software architecture, we can obtain a more detailed metric: $M_S' = max\{|\sigma_{[1]=v}(D_t^+)| \mid v \text{ is a vertex of the ADG }\}$.

As we observed, all the architectural metrics defined above are absolute metrics. In general, the larger is a architectural metric of a software architecture, the more complex is the software architecture. Moreover, some relative architectural metrics should also by considered since they can measure the complexity of a software architecture from some different viewpoints.

4 Related Work

Although much work has been studied for code metrics at implementation code level, the study of architectural metrics has not received as much as attention in comparing with code metrics. Among existing architectural metrics, there are two famous architectural metrics that have been proposed by Yin and Winchester which is derived from a system's structured design chart, and by Henry and Kafura which is derived from a system's information flow. We compare their approaches with ours here.

Yin and Winchester have defined some architectural metrics based on analysis of a system's design structure chart [22]. They focused on the interface between the major levels in a large, hierarchically structure. However, the fundamental problem for Yin and Winchester's work is that their metrics were defined based on informal system's design structure charts which can only capture the flow of information across level boundaries. In contrast, our metrics are defined based on various types of architectural dependences in a software architecture that can be derived from analyzing its formal architectural specification, and therefore, can measure the architectural complexity of the system more well.

Henry and Kafura proposed some architectural metrics based on information flow of a system. Their metrics are probably the most cited architectural metrics that have been developed. The idea behind these metrics is that complexity is measured in terms of information flow, and that more complex modules in a system are those through which large amounts of information flow. Their approach is much more detailed compared with Yin and Winchester's work because it observes all information flow rather than just flow across level boundaries. However, there are two fundamental problems in information flow metrics. First, although Henry and Kafura stated that their approach can be completely automated, this is not often the case. Recent evaluations showed that due to the ambiguous definitions of some of the metrics, it is difficult to give an evaluation of the metrics. This makes it difficulty to develop automated support tools for the approach [11, 20]. Second, information flow metrics were also defined based on some informal structure charts which usually poorly capture the attributes of a system's architecture. In contrast, our metrics which are defined in terms of various types of architectural dependences in a software architecture that can be derived from analyzing its formal architectural specification, and therefore can capture more

intrinsic and deeper attributes of a system's architecture. Moreover, due to the synthetic nature of some information flow metrics (i.e. the fact that they are obtained by combining the values of a number of other counts), recent studies showed that this may conceal underlying effects and lead to incorrect diagnoses of the status of either the system as a whole or of individual components [11]. Our metrics, in contrast, are defined based on primitive counts (of dependence arcs in the ADG), rather than synthetics, and therefore no such a problem occurred.

5 Concluding Remarks

We proposed some new architectural metrics which are appropriate for evaluating the architectural attributes of a software system. The main feature of our approach is to measure the complexity of a software architecture by capturing various types of architectural dependences in the architecture that can be derived from analyzing its formal architectural specification. In order to formally define these metrics, we presented a dependence-based representation named *Architectural Dependence Graph* (ADG) to explicitly represent these architectural dependences in the architecture.

The work presented here is primary, and there is still a lot of work that remains to be done. For example, in addition to defining metrics based on architectural dependences, similar to [2] which they defined some metrics based on program slices to evaluate functional cohesion of a program, we can also define metrics to evaluate the functional cohesion of a software architecture based on architectural slices that can be computed by a new slicing technique called *architectural slicing* [21, 23, 24]. Moreover, we can also define some architectural metrics by simply counting the number of elements in a formal architectural specification. For example, we can define metrics by counting the number of components, connections between components, and even the number of lines in a formal architectural specification. On the other hand, it is important to develop static analysis tools to automatically support the collection and evaluation of the architectural metrics proposed in this paper. Now we are implementing an architectural dependence analysis tool for WRIGHT architectural specifications and an architectural metric collector based on it. The next step for us is to perform some experiments and collect data for evaluation. We hope a primary evaluation of these metrics will be available soon.

References

[1] R. Allen, "A Formal Approach to Software Architecture," PhD thesis, Department of Computer Science, Carnegie Mellon University, 1997.

[2] J. M. Bieman and L. M. Ott, "Measuring Functional Cohesion," *IEEE Transaction on Software Engineering*, Vol.20, No.8, pp.644-657, 1994.

[3] J. Cheng, "Process Dependence Net of Distributed Programs and Its Applications in Development of Distributed Systems," *Proc. of the COMPSAC'93*, pp.231-240, 1993.

[4] N.E.Fenton and S. L. Pfleeger, "Software Metrics: A Rigorous and Practical Approach," Second Edition, International Thomson Computer Press, 1997.

[5] M. Halstead, "Elements of Software Science," Elsevier, North Holland, 1977.

[6] W. Harrison and C. Cook, "A Micro/Macro Measure of Software Complexity," *Journal of System and Software*, Vol.7, No.2, pp.213-219, 1987.

[7] S. Henry and D. Kafura, "Software Structure Measures Based on Information Flow," *IEEE Transactions on Software Engineering*, Vol.7, No.5, pp.510-518, 1981.

[8] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural Slicing Using Dependence Graphs," *ACM Transaction on Programming Language and System*, Vol.12, No.1, pp.26-60, 1990.

[9] D. C. Luckham, L. M. Augustin, J. J. Kenney, J. Veera, D. Bryan, and W. Mann, "Specification Analysis of System Architecture Using Rapide," *IEEE Transaction on Software Engineering*, Vol.21, No.4, pp.336-355, April 1995.

[10] R. Kazman and M. Burth, "Assessing Architectural Complexity," *Proceedings of the 2nd Euromicro Conference on Software Maintenance and Reengineering*, pp.104-112, Florence, Italy, March 1998.

[11] B. A. Kitchenham, L. M. Pickard, and S. J. Linkman, "An Evaluation of Some Design Measures," *Software Engineering Journal*, pp.50-58, January 1990.

[12] T. W. Malone and K. Crowston, "The Interdisciplinary Study of Coordination," *ACM Computing Surveys*, Vol.26, No.1, pp.87-119, 1994.

[13] T. J. McCabe, "A Software Complexity Measure," *IEEE Transaction on Software Engineering*, Vol.2, No.4, pp.308-320, 1976.

[14] T. J. McCabe and C. W. Butler, "Design Complexity Measurement and Testing," *Communications of the ACM*, Vol.32, No.12, pp.1415-1425, 1989.

[15] K. J. Ottenstein and L. M. Ottenstein, "The Program Dependence Graph in a software Development Environment," *ACM Software Engineering Notes*, Vol.9, No.3, pp.177-184, 1984.

[16] H. D. Rombach, "Design Measurement: Some Lessons Learned," *IEEE Software*, pp.17-25, March 1990.

[17] A. Podgurski and L. A. Clarke, "A Formal Model of Program Dependences and its Implications for Software Testing, Debugging, and Maintenance," *IEEE Transaction on Software Engineering*, Vol.16, No.9, pp.965-979, September 1990.

[18] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik, "Abstractions for Software Architecture and Tools to Support Them," *IEEE Transaction on Software Engineering*, Vol.21, No.4, pp.314-335, April 1995.

[19] M. Shaw and D. Garlan, "Software Architecture: Perspective on an Emerging Discipline," Prentice Hall, 1996.

[20] M. Shepperd, "Design Metrics: An Empirical Analysis," *Software Engineering Journal*, pp.3-10, January 1990.

[21] J. A. Stafford, D. J. Richardson, and A. L. Wolf, "Aladdin: A Tool for Architecture-level Dependence Analysis of Software Systems," Technical Report CU-CS-858-98, Department of Computer Science, University of Colorado, April 1998.

[22] B. H. Yin and J. W. Winchester, "The Establishment and Use of Measures to Evaluate the Quality of Software Designs," *Proceedings of the Software Quality and Assurance Workshop*, pp.45-52, 1978.

[23] J. Zhao, "Using Dependence Analysis to Support Software Architecture Understanding," in M. Li (Ed.), *New Technologies on Computer Software*, pp.135-142, International Academic Publishers, September 1997.

[24] J. Zhao, " Applying Slicing Technique to Software Architectures," *Proceedings of the Fourth IEEE International Conference on Engineering of Complex Computer Systems*, pp.87-98, Monterey, USA, August 1998.

[25] H.Zuse, "*Software Complexity: Measures and Methods*," Walter de Gruyter, 1990.