Towards A Metrics Suite for Aspect-Oriented Software

Jianjun Zhao
Department of Computer Science and Engineering
Fukuoka Institute of Technology
3-30-1 Wajiro-Higashi, Higashi-ku, Fukuoka 811-0295, Japan
zhao@cs.fit.ac.jp

ABSTRACT

Although a large body of research in software metrics has been focused on procedural or object-oriented software, there is no software metric for aspect-oriented software until now. In this paper, we propose some metrics for aspect-oriented software, which are specifically designed to quantify the information flows in an aspect-oriented program. We define these metrics based on a dependence model for aspect-oriented software which consists of a group of dependence graphs each can be used to explicitly represent various dependence relations at different levels of an aspect-oriented program. The proposed metrics can be used to measure the complexity of an aspect-oriented program from various different

1. INTRODUCTION

Software metrics aim to measure the inherent complexity of software systems with a view toward predicting the overall project cost and evaluating the quality and effectiveness of the design. Software metrics have many applications in software engineering tasks such as program understanding, testing, refactoring, maintenance, and project management.

Research for software measurement must adapt to the emergence of new software development methods, and metrics for new languages and design paradigms must be defined based on models that are relevant to these new paradigms [3].

Recently, aspect-oriented programming has been proposed as a technique for improving separation of concerns in software design and implementation [9, 12]. Aspect-oriented programming works by providing explicit mechanisms for capturing the structure of crosscutting concerns in software systems. Aspect-oriented programming languages can be used to cleanly modularize the crosscutting structure of concerns such as exception handling, synchronization, performance optimizations, and resource sharing, that are usually difficult to express cleanly in source code using existing programming techniques. Aspect-oriented programming languages can control such code tangling and make the underlying concerns more apparent, making programs easier to develop and maintain. As research in aspect-oriented programming is reaching maturity with a number of active research products, software metrics researchers need to focus on this new paradigm in order to efficiently evaluate it in a rigorous and quantitative fashion.

In an aspect-oriented system, the basic program unit is an aspect rather than a procedure or a class. An aspect with its encapsulation of state with associated advice (operations) is a significantly different abstraction than the procedure units within procedural programs or class units within object-oriented programs. The inclusion of join points in an aspect further complicates the static relations among aspects and classes. Therefore, in order to define some metrics for estimating the complexity of aspect-oriented software, models that are appropriate for representing aspect-oriented systems are needed.

However, although a large body of research in software metrics has been focused on procedural or object-oriented software [3, 4, 5, 6, 7, 11, 16] as well as software architectures [8, 13, 14], until now there is no software metric for aspect-oriented software. Further, due to the specific features of aspect-oriented software, existing models and abstractions for procedural or object-oriented software can not be applied to aspect-oriented software straightforwardly.

In this paper, we propose some metrics for assessing the complexity of aspect-oriented software, which are specifically designed to quantify the information flows in an aspect-oriented program. These metrics are defined based on a dependence model for aspect-oriented software which consists of a group dependence graphs defined at three levels of an aspect-oriented program to explicitly represent various dependence relations in the program. The proposed metrics can be used to measure the complexity of aspect-oriented software from various different viewpoints.

The rest of the paper is organized as follows. Section 2 gives some background information related to this research. Section 3 presents dependence graphs for aspect-oriented software at the three levels of abstraction. Section 4 defines a set of complexity metrics for aspect-oriented programs based on these dependence graphs, and concluding remarks are given in Section 5.

Technical-Report SE-136-25, Information Processing Society of Japan (IPSJ), March 2002.

```
 ce0 public class Point {
  s1   protected int x, y;
 me2   public Point(int _x, int _y) {
  s3     x = _x;
  s4     y = _y;
       }
 me5   public int getX() {
  s6     return x;
       }
 me7   public int getY() {
  s8     return y;
       }
 me9   public void setX(int _x) {
 s10     x = _x;
       }
me11   public void setY(int _y) {
 s12     y = _y;
       }
me13   public void printPosition() {
 s14     System.out.println("Point at("+x+","+y+")");
       }
me15   public static void main(String[] args) {
 s16     Point p = new Point(1,1);
 s17     p.setX(2);
 s18     p.setY(2);
       }
     }

ce19 class Shadow {
 s20   public static final int offset = 10;
 s21   public int x, y;

me22   Shadow(int x, int y) {
 s23     this.x = x;
 s24     this.y = y;
me25   public void printPosition() {
 s26     System.outprintln("Shadow at
                ("+x+","+y+")");
       }
     }
```

```
ase27 aspect PointShadowProtocol {
  s28   private int shadowCount = 0;
 me29   public static int getShadowCount() {
  s30     return PointShadowProtocol.
                 aspectOf().shadowCount;
       }
  s31   private Shadow Point.shadow;
 me32   public static void associate(Point p, Shadow s){
  s33     p.shadow = s;
       }
 me34   public static Shadow getShadow(Point p) {
  s35     return p.shadow;
       }

 pe36   pointcut setting(int x, int y, Point p):
          args(x,y) && call(Point.new(int,int));
 pe37   pointcut settingX(Point p):
          target(p) && call(void Point.setX(int));
 pe38   pointcut settingY(Point p):
          target(p) && call(void Point.setY(int));

 ae39   after(int x, int y, Point p) returning :
            setting(x, y, p) {
  s40     Shadow s = new Shadow(x,y);
  s41     associate(p,s);
  s42     shadowCount++;
       }
 ae43   after(Point p): settingX(p) {
  s44     Shadow s = new getShadow(p);
  s45     s.x = p.getX() + Shadow.offset;
  s46     p.printPosition();
  s47     s.printPosition();
       }
 ae48   after(Point p): settingY(p) {
  s49     Shadow s = getShadow(p);
  s50     s.y = p.getY() + Shadow.offset;
  s51     p.printPosition();
  s52     s.printPosition();
      }
    }
```

Figure 1: A sample AspectJ program.

2. ASPECT-ORIENTED PROGRAMMING WITH ASPECTJ

We assume that readers are familiar with the basic concepts of aspect-oriented programming, and in this paper, we use AspectJ [1] as our target language to show the basic idea of our metrics for aspect-oriented software. The selection of AspectJ is based on that it is one of most popular aspect-oriented language in the community.

Below, we use a sample program taken from [1] to briefly introduce the AspectJ. The program shown in Figure 1 associates shadow points with every `Point` object and contains one `PointShadowProtocol` aspect that stores a shadow object in every `Point` and two classes `Point` and `Shadow`.

AspectJ is a seamless aspect-oriented extension to Java. AspectJ adds some new concepts and associated constructs to Java. These concepts and associated constructs are called join points, pointcut, advice, introduction, and aspect.

The *join point* is an essential element in the design of any aspect-oriented programming language since join points are the common frame of reference that defines the structure of crosscutting concerns. The join points in AspectJ are well-defined points in the execution of a program. The join points in AspectJ are *method* or *constructor call*, *method* or *constructor execution*, *class* or *object initialization*, *field reference* or *assignment*, and *handler execution* [1].

A *pointcut* is a set of join points that optionally exposes some of the values in the execution of those join points. AspectJ defines several primitive *pointcut designators* that can identify all types of join points. Pointcuts in AspectJ can be composed and new pointcut designators can be defined according to these combinations. For example, In aspect `PointShadowProtocol` three pointcuts are declared with the

names of `setting`, `settingX`, and `settingY`.

Advice is used to define some code that is executed when a pointcut is reached. Aspectj provides three types of advice, that is, *before*, *after*, and *around*. In addition, there are also two special cases of *after* advice, called *after returning* and *after throwing*. For example, In aspect `PointShadowProtocol`, there are three pieces of *after* advice `setting`, `settingX`, and `settingY`.

Advice declarations can change the behavior of classes they crosscut, but can not change their static type structure. For crosscutting concerns that can operate over the static structure of type hierarchies, AspectJ provides forms of introduction.

Introduction in AspectJ can be used by an aspect to add new fields, constructors, or methods (even with bodies) into given interfaces or classes. Introduction can be public or private, where a private introduction means only code in the aspect that declared it can refer or access the introduced fields, constructors, or methods. For example, In aspect `PointShadowProtocol`, introduction declaration `private Shadow Point.shadow;` privately introduces a field named `shadow` of type `Shadow` in `Point`. This means that only code in the aspect can refer to `Point`'s `shadow` field.

Aspects are modular units of crosscutting implementation. Aspects are defined by aspect declarations, which have a similar form of class declarations. Aspect declarations may include advice, pointcut, and introduction declarations as well as other declarations such as method declarations, that are permitted in class declarations. For example, the program in Figure 1 defines one aspect named `PointShadowProtocol`.

An AspectJ program can be divided into two parts: *non-aspect code* which includes some classes, interfaces, and other language constructs as in Java, and *aspect code* which includes aspects for modeling crosscutting concerns in the program. For example, the sample program of Figure 1 can be divided into the non-aspect code containing classes `Point` and `Shadow`, and the aspect code which has aspect `PointShadowProtocol`. Moreover, any implementation of AspectJ is to ensure that the aspect and non-aspect code run together in a properly coordinated fashion. Such a process is called *aspect weaving* and involves making sure that applicable advice runs at the appropriate join points. For detailed information about AspectJ, one can refer to [1].

3. A DEPENDENCE MODEL FOR ASPECT-ORIENTED SOFTWARE

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. Aspect-oriented programming languages differ from procedural or object-oriented programming languages in many ways. Some of these differences, for example, are the concepts of joint points, advice, aspects, and their associated constructs. These aspect-oriented features may impact on the development of model for aspect-oriented software.

In this section, we present a dependence model for aspect-oriented software to capture attributes concerning about information flow in an aspect-oriented program. The model consists of dependence graphs representing an aspect-oriented system at three levels, i.e., the module-level, aspect-level, and system-level. Each level has its own dependence graph, and therefore support to develop dependence-based metrics at different levels.

3.1 Module-Level Dependence Graphs

In this subsection, we describe how to represent a module, i.e., a piece of advice, an introduction, or a method in an aspect using a dependence graph.

We use the *method dependence graph* (MDG) to represent a method in an aspect. The MDG is a digraph whose vertices represent statements or predicate expressions in the method and arcs represent two types of dependence relationships, i.e., *control dependence*, and *data dependence*. Control dependence represents control conditions on which the execution of a statement or expression depends in the method. Data dependence represents the data flows between statements in the method. Each MDG has a unique vertex called *method start vertex* to represent the entry of the method.

We use the *advice dependence graph* (ADG) to represent a piece of advice in an aspect. The ADG is similar to the MDG of a method such that its vertices represent statements or predicate expressions in the advice, and its arcs represent control or data dependencies between vertices. Each piece of advice has a unique vertex called *advice start vertex* to represent the entry of the advice.

We use the *introduction dependence graph* (IDG) to represent an introduction in an aspect. The IDG of an introduction is similar to the MDG of a method such that its vertices represent statement or predicate expressions in the introduction and its arcs represent control or data dependencies between these statements. There is a unique vertex called *introduction start vertex* in the IDG to represent the entry of the introduction.

3.2 Aspect-Level Dependence Graphs

We use the *aspect interprocedural dependence graph* (AIDG) to represent a single aspect in an aspect-oriented program.

The AIDG of an aspect is a digraph that consists of a number of ADGs, IDGs, and MDGs each representing a piece of advice, an introduction, or a method in the aspect, and some special kinds of dependence arcs to represent direct or indirect dependencies between a call and the called advice, introduction, or method and transitive interprocedural data dependencies in the aspect. Each AIDG has a unique vertex called *aspect start vertex* to represent the entry into the aspect. The aspect start vertex is connected to each start vertex of an ADG, IDG, or MDG in the aspect by *aspect membership arcs* to represent the membership relations.

In order to model parameter passing in an aspect. Formal-in and formal-out vertices are associated with each advice, introduction, or method start vertex, and actual-in and actual-out vertices are associated with each call vertex representing a call site in the aspect. Each formal parameter vertex is

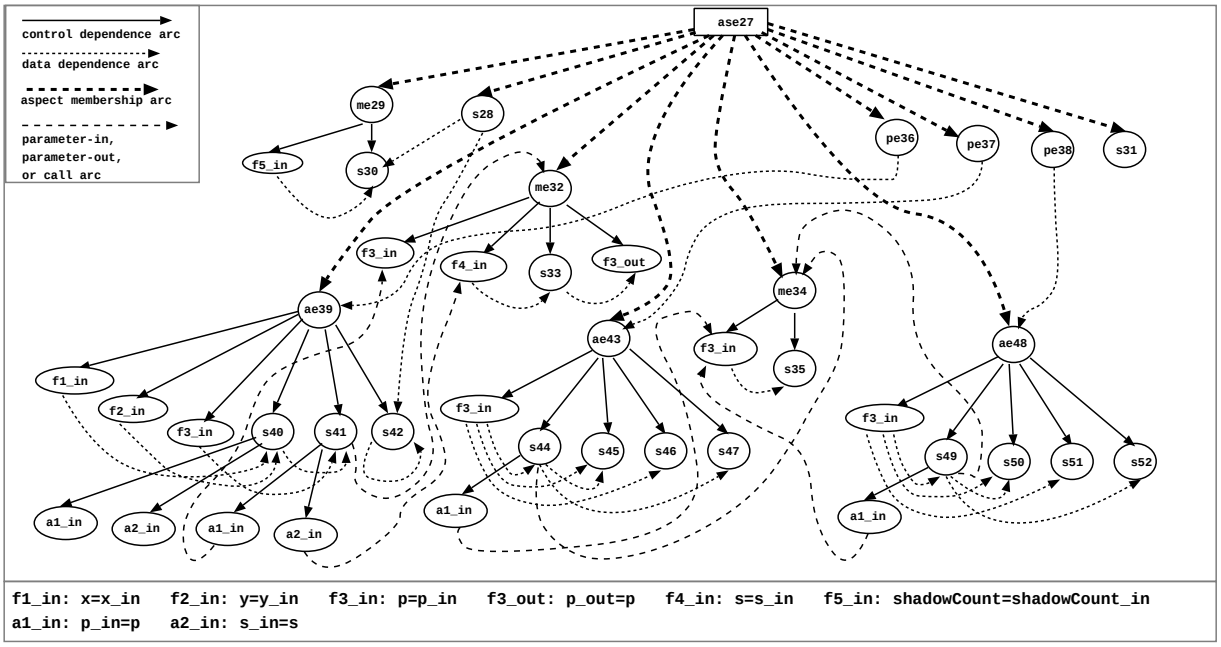

Figure 2: An AIDG for aspect *PointShadowProtocol* of the program in Figure 1.

control-dependent on the start vertex, and each actual parameter vertex is control-dependent on the call vertex.

For the instance variables declared in an aspect, since they are accessible to all advice, introductions, and methods in the aspect, we create formal-in and formal-out vertices for all instance variables that are referenced in the advice, introductions, and methods.

Finally, for each pointcut, we connect the aspect start vertex to each pointcut start vertex through aspect membership arcs, and also connect each pointcut start vertex to its corresponding advice start vertex by call dependence arcs to represent relationships between them.

Example. Figure 2 shows the AIDG of aspect `PointShadowProcotol`. For example, in the figure, $ase27$ is the aspect start vertex, and $ae39$, $pe36$, $ie31$ and $me32$ are advice, pointcut, introduction, and method start vertices respectively. $(ase27, ae39)$, $(ase27, pe36)$, $(ase27, ie31)$, and $(ase27, me32)$ are aspect membership arcs. Moreover, each advice, introduction, or method start vertex is the root of a sub-graph which is itself an ADG, IDG, or MDG.

3.3 System-Level Dependence Graph

We present a system-level dependence graph called the *aspect-oriented system dependence graph* (ASDG) to represent a complete aspect-oriented program. An ASDG of an aspect-oriented program is a collection of dependence graphs each representing a piece of advice, an introduction, or a method in an aspect of the program, and some additional arcs to represent direct or indirect dependencies between a call and the called module and some transitive interprocedural data dependencies. We first introduce how to represent interactions among aspects and classes, and then construct the complete ASDG.

3.3.1 Representing Interactive Aspects and Classes

An aspect can interact with a class by four ways: (1) creating an object of the class from the aspect, (2) there is a call from a method or a piece of advice in the aspect to a method of the class, (3) declaring a public introduction in the aspect to add a field, method, or constructor to the class, and (4) weaving the code declared in advice of the aspect to the class code at join points. Where the first and the second ways are similar to class interactions in an object-oriented program, the third and the fourth ways are unique for aspect-oriented programs. In the following, we describe how to present these four interactions.

Creating Objects

In AspectJ, an aspect may create an object of a class through a declaration or by using an operator such as `new` similar to a Java class. When an aspect A creates an object of class C, there is an implicit call to C's constructor. To represent this implicit constructor call, we add a call vertex in A at the place of object creation. A call dependence arc connects this call vertex to the start vertex of the C's constructor MDG. In the meantime, actual-in and actual-out vertices are added at the call vertex to match the formal-in and formal-out vertices in C's constructor MDG.

Making Calls

When there is a call site in method m_1 or advice a_1 in aspect A to method m_2 in the public interface of C, we connect the call vertex of m_1 in A to the method start vertex of m_2 to form a call dependence arc, and also connect actual-in and formal-in vertices to form parameter-in dependence arcs and

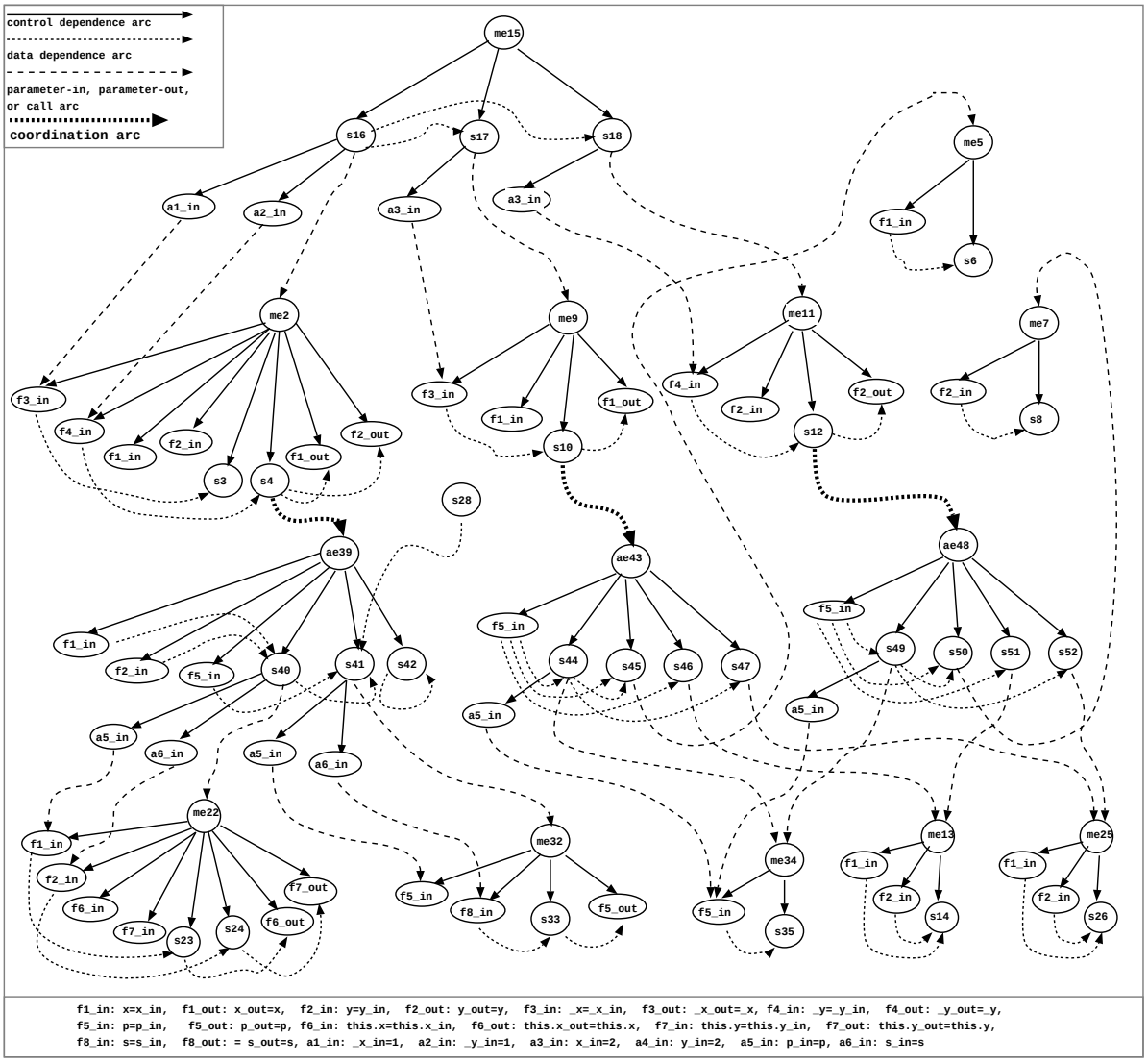

Figure 3: An ASDG of the program in Figure 1.

actual-out and formal-out vertices to form parameter-out dependence arcs.

Using Introductions

In AspectJ, an aspect A can also interact with a class C by declaring a public introduction I in A for adding an additional field, method, or constructor to C. To represent such an interaction, we connect the class start vertex of C's class dependence graph to the introduction start vertex of the I's IDG by a class membership arc.

Using Join Points

In AspectJ, join points are defined in each aspect with the *pointcut* designator. Pointcuts are further used in the definition of *advice*. By carefully examining join points declared in the pointcuts and their associated advice, one can determine the weaving points statically in the non-aspect code to facilitate the connection of the non-aspect code to the aspect code. In this paper, we use weaving vertices in the SDG to represent the weaving points in the non-aspect code which can be used to connect the SDG of non-aspect code to the AIDGs of aspect code.

For example, in order to determine the weaving point for weaving the code declared in advice `settingY` to a method in class `Point`. First, from pointcut `settingY` declaration, we knew that the code in advice `settingY` should be inserted into method `setY` of class `Point`. However, we still do not know the exact place where we should insert the code. By examining advice `settingY`'s declaration we further know that this advice is *after* advice. According to the AspectJ programming guide [1]: "*after advice runs after the computation 'under the join point' finishes, i.e., after the method body has run, and just before control is returned to the caller*

(p.12)," we know that the code declared in advice `settingY` should be inserted into the place after the last statement of method `setY`, i.e., after `y = _y`. Similarly, we can determine other weaving points in the non-aspect code.

3.3.2 Aspect-Oriented System Dependence Graph

Generally, an AspectJ program consists of classes, interfaces, and aspects. In order to execute the program, the program must include a special class called `main()` class. The program first starts the `main()` class, and then transfers the execution to other classes.

To construct the ASDG for a complete AspectJ program, we first construct the SDG for the non-aspect code using existing techniques proposed for object-oriented programs [10, 15] and then insert the weaving vertices to the SDG. After that, we use a *coordination dependence arc* to connect each weaving vertex to the advice start vertex of its corresponding ADG. A call dependence arc is added between a call vertex and the start vertex of the ADG, IDG, or MDG of the called advice, introduction, or method. Actual and formal parameter vertices are connected by parameter dependence arcs.

Example. Figure 3 shows the complete ASDG of the sample AspectJ program in Figure 1.

4. METRICS FOR ASPECT-ORIENTED SOFTWARE

Since program dependencies are dependence relationships holding between program elements in a program that are determined by control flows and data flows in the program, they can be regarded as one of intrinsic attributes of programs. Therefore it is reasonable to take program dependencies as one of objects for measuring program complexity.

In this section, we define a set of complexity metrics in terms of program dependence relations to measure the complexity of an aspect-oriented program from various viewpoints. Once the dependence graphs of an aspect-oriented program is constructed, the metrics can be easily computed in terms of dependence graphs.

4.1 Module-Level Metrics

We first define some metrics at the module level based on the ADG, IDG, and MDG. These metrics can be used to measure various complexities of a piece of advice, an introduction, or a method from a general viewpoint. Let α be a module, i.e., a piece of advice, an introduction, or a method in an aspect and G_α be the dependence graph of α, we have the following metrics.

- $\mathbf{M_1}$: the number of all control dependence arcs in G_α. It can be used to measure the complexity of a module from a special viewpoint of control structure.
- $\mathbf{M_2}$: the number of all data dependence arcs in G_α. It can be used to measure the complexity of a module from a special viewpoint of information flow.
- $\mathbf{M_3}$: the number of all program dependence arcs in G_α. It can be used to measure the total complexity of a module from a general viewpoint.

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

- $\mathbf{M_4}$: the maximal number of vertices that a vertex is somehow dependent on G_α. It can be used to determine the "most easily affected" statement(s) in a module.

Since all the metrics defined above are absolute metrics, they have the following property:

- *In general, the larger is the value of a metric of α, the more complex is α.*

4.2 Aspect-Level Metrics

We can also define some aspect-level metrics for an individual aspect based on its AIDG. These metrics can be used to measure various complexities of an aspect from different viewpoints. Let β be an aspect and G_β be the AIDG of β, we have the following metrics.

- $\mathbf{M_5}$: the number of all control dependence arcs in G_β. It can be used to measure the complexity of an aspect from a special viewpoint of intraprocedural control structure.
- $\mathbf{M_6}$: the number of all data dependence arcs in G_β. It can be used to measure the complexity of an aspect from a special viewpoint of intraprocedural information flow.
- $\mathbf{M_7}$: the number of all call dependence arcs in G_β. It can be used to measure the complexity of an aspect from a special viewpoint of interprocedural control structure.
- $\mathbf{M_8}$: the number of all parameter-in and parameter-out dependence arcs in G_β. It can be used to measure the complexity of an aspect from a special viewpoint of interprocedural information flow.
- $\mathbf{M_9}$: the number of all call, parameter-in, and parameter-out dependence arcs in G_β. It can be used to measure the complexity of an aspect from a special viewpoint of interprocedural control structure and information flow.
- $\mathbf{M_{10}}$: the number of all control, data, call, parameter-in, parameter-out dependence arcs in G_β. It can be used to measure the total complexity of an aspect from a general viewpoint.
- $\mathbf{M_{11}}$: the maximal number of vertices that a vertex is somehow dependent on in G_β. It can be used to determine the "most easily affected" statement(s) in an aspect.

Note that according to the above definitions, we can get that $\mathbf{M_9} = \mathbf{M_7} + \mathbf{M_8}$ and $\mathbf{M_{10}} = \mathbf{M_5} + \mathbf{M_6} + \mathbf{M_7} + \mathbf{M_8}$. Moreover, since all the metrics defined above are absolute metrics, they have the following property:

- *In general, the larger is the value of a metric of β, the more complex is β.*

4.3 System-Level Metrics

Finally, we can define some metrics at the whole system level based on the ASDG. These metrics can be used to measure various total complexities of an aspect-oriented program from various viewpoints. Let ρ be an aspect-oriented program and G_ρ be the ASDG of ρ, we have the following metrics:

- $\mathbf{M_{12}}$: the number of all control, call, and coordination dependence arcs in G_ρ. It can be used to measure the total complexity of an aspect-oriented program from a special viewpoint of control structure.
- $\mathbf{M_{13}}$: the number of all data, parameter-in, and parameter-out dependence arcs in G_ρ. It can be used to measure the total complexity of an aspect-oriented program from a special viewpoint of information flow.
- $\mathbf{M_{14}}$: the number of all dependence arcs in G_ρ. It can be used to measure the total complexity of an aspect-oriented program from a general viewpoint.
- $\mathbf{M_{15}}$: the maximal number of vertices that a vertex is somehow dependent on in G_ρ. It can be used to determine the "most easily affected" statement(s) in an aspect-oriented program.

Note that according to the above definitions, we can get that $\mathbf{M_{14}} = \mathbf{M_{12}} + \mathbf{M_{13}}$. Moreover, since all the metrics defined above are absolute metrics, they have the following property:

- *In general, the larger is the value of a metric of ρ, the more complex is ρ.*

5. CONCLUDING REMARKS

In this paper, we proposed some metrics for aspect-oriented software, which are specifically designed to quantify the information flows of aspect-oriented programs. These metrics are defined based on the dependence model of aspect-oriented software which consists of dependence graphs defined at three levels to explicitly represent various dependencies in an aspect-oriented program. The proposed metrics can be used to measure the complexity of aspect-oriented software from various different viewpoints.

While our initial exploration used AspectJ as our target language, the concept and approach presented in this paper are language independent. However, the implementation of a dependence analysis tool may differ from one language to another because each language has its own structure and syntax that must be handled appropriately. As one of our future work, we plan to develop a dependence analysis tool for AspectJ which includes a generator for automatically constructing different levels of dependence graphs for AspectJ programs and a metrics collection tool for collecting and evaluation of the metrics presented in this paper of an AspectJ program based on these dependence graphs.

6. REFERENCES

[1] The AspectJ Team, "The AspectJ Programming Guide," 2001.

[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. M. Bieman, "Metrics Development for Object-Oriented Software," In *Software Measurement: Understanding Software Engineering*, A. Melton, editor, International Thomson Publishing (ITP), pp.75-93, 1996.

[4] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object-Oriented Design," *IEEE Transaction on Software Engineering*, Vol.20, No.6, pp.476-498, 1994.

[5] J. Cheng, "Complexity Metrics for Distributed Programs," *Proc. the 4th IEEE International Symposium on Software Reliability*, pp.132-141, Denver, U.S.A., November 1993.

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

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

[8] 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.

[9] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," *proc. 11th European Conference on Object-Oriented Programming*, pp220-242, LNCS, Vol.1241, Springer-Verlag, June 1997.

[10] L. D. Larsen and M. J. Harrold, "Slicing Object-Oriented Software," *Proceeding of the 18th International Conference on Software Engineering*, German, March, 1996.

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

[12] P. Tarr, H. Ossher, W. Harrison, and S. Sutton, "N Degrees of Separation: Multi-Dimensional Separation of Concerns," *Proc. the 21th International Conference on on Software Engineering*, pp.107-119, 1999.

[13] 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.

[14] J. Zhao, "On Assessing the Complexity of Software Architectures," *Proc. 3rd International Software Architecture Workshop*, pp.163-166, ACM SIGSOFT, November 1998.

[15] J. Zhao, "Slicing Concurrent Java Programs," *Proc. Seventh IEEE International Workshop on Program Comprehension*, pp.126-133, May 1999.

[16] H.Zuse, "*A Framework of Software Measurement*," Walter de Gruyter, 1997.