

A Graph-Based Representation of Object-Oriented Designs

Wei Li, Huaming Zhang, and Raed Shatnawi
Computer Science Department
The University of Alabama in Huntsville
Huntsville, AL, 35899

Abstract

This article proposes a general graph-based representation for modeling Object Oriented (OO) software designs. The advantages of the representation over the Unified Modeling Language (UML) are: it formalizes the link between primitive OO design features and primitive entities of graphs; it models all the design features of an OO design in one cohesive graph; it provides us a convenient platform for formal analysis and reasoning. Thus, the representation allows us to investigate various properties of OO designs, through their graph representations. These properties include software metrics, link predications and small world phenomenon etc.

Keywords: Object-Oriented Design and Modeling, Graph Representation

1 Introduction

Graphs are often used to model entities and their relations. The Unified Modeling Language (UML) is perhaps the most commonly used graph-based representation for OO design. UML is a diagram-based OO design language that comprises the use case diagram, class diagram, object diagram, interaction diagram, component diagram, collaboration diagram, activity diagram, deployment diagram, and state transition diagram [OMG 2005]. UML is a powerful and comprehensive OO design language. It provides good visualization and comprehensive modeling for OO designs, but it tends to scatter a design into many diagrams to visualize different design views. Therefore, it is not a convenient platform for formal design analysis and reasoning because the graph notations are not uniform. This inconvenience prompted us to seek a new graph representation for object-oriented designs.

2 Big-Bang Graph Representation of OO designs

We name our graph of modeling the OO designs big-bang graph (BBG), it is a colored, directed, and connected graph, which is denoted by $BBG=(V, E)$, where V is a set of colored vertices and E is a set of colored and directed edges. The vertices of BBG are used to represent design entities such as use cases, use-case realization, actors, concerns, aspects, join points, point cuts, classes, objects, object references, methods, procedures/functions, sequential statements, decision statements, and the dot—the execution starting point of the system. The edges are used to represent the primitive associations of the design entities. The goal of BBG is to model software design that includes the most primitive (atomic) design features in one simple graph notation. A *primitive design feature* is a design notion which is irreducible (it cannot be further divided) in a particular paradigm. For example, the *object read* is a primitive design feature in the OO paradigm, whereas *object access* is not, because it can be further divided into two primitive features: *object read* and *object write*. Note that, we can make all edges, but not all vertices, primitive for practical concerns. For example, the concept of a *class* is not primitive because it is composed of *attributes* and *operations*. Yet, we cannot represent attributes and operations without modeling classes in BBG. Therefore, it is sufficient

to say that all the edges in BBG represent the primitive design features. As we can see, BBG is a generic design representation and is capable of modeling the aspect-oriented, object-oriented, and procedural design, etc. For brevity, we focus only on the part of BBG that is used to model OO designs in this article. Figure 1 summarizes the BBG vertices and their colors. In the model, the black dot models the entry point of the program, which is the main function if we use Java.

Node type	Node color
Method	⦿ (black) and ● (The Dot)
Class	⦿ (red)
Object Reference	⦿ (blue)
Object	⦿ (cyan)
System	⦿ (rosa)

Figure 1: BBG Vertices

We will treat all variables as objects when we model OO designs by using BBG. For those variables of the traditional primitive types—char, int, and float, etc—that are popular in C++ and Java, we abstract them as objects in BBG. In addition, we add a System vertex, which is the abstraction of all possible design entities that are outside the scope of a design.

After modeling design entities by vertices in BBG, we represent the vertex-association semantic relation such as definition, reading from, writing to, and message passing, as edges in BBG. Each edge has a unique color which is based on the color of the source vertex. Figure 2 summaries the edges.

It is critical to model only the primitive (atomic) design features as the colored edges in BBG. For example, the *object read* and *object write* are modeled as two different colored edges in BBG because breaking down either one would not make sense from the OO design perspective. However, *object access* is not modeled as an edge in BBG because it can be *derived* from two primitive design features: *object read* and *object write*. There are several benefits of modeling only the primitive design features as edges. First, the model guarantees that each edge color is associated with a single primitive design concept. Thus each edge color represents a homogeneous relation on the vertex set. Second, as described fully in [Li and Zhang 2006], the software metrics that is characterized by a set of single-colored edges is guaranteed to be an atomic metrics that measures a primitive design concept.

Note that, when using BBG to model OO designs, the evolution of an OO design is just the refinement of the finite vertices and edge relations in the design.

There should be one axiomatic definition for each relation in Figure 2 (we omit the complete list of definitions for brevity).

The set of edges—E—in BBG is the union of all the semantic relations in Figure 2. When BBG is expanded to model more design features, E will include more edges pertaining to the expanded design features.

We would like to point out some assumptions that we made about the semantic relation definitions. First, we define any semantics that ends in an object to the object’s reference that points to the object. Second, although the relations are defined using object references, we assume that all actions to an object reference will take place in the object that is referred to (pointed to) by the object reference. For example, if a method writes to an object reference, the writing will take place in the object that the reference points to. If an object reference points to several objects, which object will receive the action is nondeterministic. Third, because all design relations to an object go through the objects reference in our model, we use the type of an object reference as the type of the object in the definitions, although the type of the object reference and the type of the object may be different because of polymorphism. We now illustrate the BBG representation through an example.

Node type	Edge color	Connection node	Edge semantics
Method	→ (black1)	System	Method to System message
	→ (black2)	Method	Method to method message
	→ (black3)	Method	Instance method overriding
	→ (black4)	Method	Class method overriding
	→ (black5)	Object reference	Being invoked through object reference
	→ (black6)	Class	Being invoked through class
	→ (black7)	Class	Returning type from method design
	→ (black8)	Object reference	Object read
	→ (black9)	Object reference	Object write
Class	→ (red1)	Class	Containment
	→ (red2)	Class	Aggregation
	→ (red3)	Class	Association
	→ (red4)	Class	Dependency
	→ (red5)	Class	Inheritance
	→ (red6)	Interface	Implementation
	→ (red7)	Object reference	Private instance attribute declaration
	→ (red8)	Object reference	Public instance attribute declaration
	→ (red9)	Object reference	Private class attribute declaration
	→ (red10)	Object reference	Public class attribute declaration
	→ (red11)	Method	Private instance method definition
	→ (red12)	Method	Public instance method definition
	→ (red13)	Method	Private class method definition
	→ (red14)	Method	Public class method definition
	→ (red15)	Method	Public instance final method definition
	→ (red16)	Method	Public class final method definition
Object Reference	→ (blue1)	Method	Visibility through parameter
	→ (blue2)	Method	Visibility through class attribute
	→ (blue3)	Method	Visibility through inherited attribute
	→ (blue4)	Method	Visibility through return
	→ (blue5)	Method	Visibility through local variable
	→ (blue6)	Object reference	Being read through object reference
	→ (blue7)	Object reference	Being written through object reference
	→ (blue8)	Class	Being read through class
	→ (blue9)	Class	Being written through class
	→ (blue10)	Class	Object reference type
	→ (blue11)	Object reference	Assignment by equality
	→ (blue12)	Object reference	Binding by value
	→ (blue13)	Object reference	Binding by reference
	→ (blue14)	Object	Referring to
Object	→ (cyan1)	Class	Object type
	→ (cyan2)	Method	Object returned from method invocation
	→ (cyan3)	System	Object returned from system
System	→ (green1)	Method	System to method message

Figure 2: BBG Edges

3 The Example

When we use BBG to model an object-oriented design, we first shall solve the duplicate name problem if it exists in an entity category—class, object, object reference, or method, etc. For example, if we define a method *m* in both class *c1* and *c2*, we can use *c1.m* and *c2.m* to denote the two different methods in our model. We treat all variable names as object references which refer to (point to) objects. The object names are generated and assigned to objects automatically by the BBG modeler; we can imagine the modeler as an automaton that generates the BBG design. For example, the first object and the second object in a design are named as *o1* and *o2* by the modeler respectively. For the object references that are not assigned names by the designer, the modeler also assigns names for them, i.e. *or1*, *or2*, etc.

We show a sample program in Java, and the BBG modeling of the design. In the BBG modeling of the design, we omitted the *ObjectType* (except the *o14-int* edge for illustration purpose), *ObjectReferenceType* (except the *args-String[]* edge for illustration purpose), and the *VisibilityThroughAttribute* edges, because that the large number of the edges would have made the drawing unwieldy. We only drew some representative objects, object references, and their perspective edges because modeling all of them would have crowded the drawing too much. We did not model anything that was related to the System except the messages. One thing we would like to point out is that although BBG can be used to visualize OO designs, its primary purpose is to provide a platform for formal design modeling and reasoning.

The following is the sample Java program:

```
class Account {
    private float balance; private String accountNumber;
    public Account() {balance=0;}
    public Account(float initialBalance) {balance=initialBalance; }
    public float addFund(float inFund) {
        balance=balance+inFund;
        return balance; }
    public String getAccountNumber () {return accountNumber;}
    public float verifyBalance(float amount)
        {if (balance > amount) return amount; else return -1; }
    public float viewAccount() {return balance; }
    public float withdrawFund(float outFund)
        {balance=balance-outFund; return balance; }
}
class BankService{
    private Customer aCustomer;
    public BankService(String userName, String password){
        if (validateUser(userName, password))
            aCustomer = new Customer(userName, password);
    }
    public Customer getCustomer(){return aCustomer; }
    private boolean validateUser(String uname, String psword) {return true; }
}
class Check {private int checkNumber; } // This is a stub class.
class CheckingAccount extends Account {
```

```

    private Check[] checks;
    public Check[] getChecks() {return checks;}
}
class CreditCardAccount extends Account { }
class Customer {
    private String name;
    private String pword;
    private CheckingAccount checkingAccount;
    private CreditCardAccount creditCardAccount;
    public Customer (String aName, String aPassword) {
        name = aName; pword = aPassword;
        checkingAccount = new CheckingAccount();
        creditCardAccount = new CreditCardAccount();
    }
    public CheckingAccount getCheckingAccount() { return checkingAccount;}
    public CreditCardAccount getCreditCardAccount() {
        return creditCardAccount;}
    public String getName () {return name;}
}
class Driver{
    public static void main(String[] args) throws Exception{
        BankService service = new BankService("aUser", "aPassword");
        CheckingAccount anAccount = service.getCustomer().getCheckingAccount();
        int userSelection;
        do {
            System.out.println("Please make a selection (0, 1, 2, or 3)");
            userSelection = System.in.read();
            switch(userSelection) {
                case '1': printToScreen(anAccount.viewAccount()); break;
                case '2': printToScreen(anAccount.addFund(10)); break;
                case '3': printToScreen(anAccount.withdrawFund(2)); break;
            }
        } while ((char) (userSelection) != '0');
    }
    private static void printToScreen(float f){System.out.println(f); }
}

```

Figure 3 illustrates the BBG representation of the program.

4 Applications of the BBG Representation Framework

The application of the BBG representation framework on software metrics and its relation with measurement theory have been thoroughly investigated in [Li and Zhang 2006]. In addition, the BBG representation can tie the aspect-oriented, object-oriented, and procedural designs together with added vertices and edges. It can also tie different models (requirements, analysis, design, and implementation) within one design paradigm together in a single graph notation. Jacobson and Ng proposed the use of *use-case slices* as the modeling unit to keep cross-cutting concerns separate in all the models—requirements, analysis, design, and implementation [Jacobson and Ng 2005]. BBG can readily model the use-case slices with additional vertices and edges, thus making it a convenient tool for linking various models in one cohesive graph representation.

Another contribution of the BBG platform is that we can create a state space for the OO design (it can create the state space for any design paradigm granted that the design paradigm is modeled properly in BBG). The state space for the OO design is represented by the basic sets that were defined at the beginning of the article.

With one single graph notation for various OO design views, it is now possible to investigate various graph phenomena in the entire graph or in a subgraph. We can single out many subgraphs from BBG. Each set of single-color edges forms its own subgraph in BBG, which can be used to study a particular aspect of the design. For example, all the *black2* edges form a subgraph in BBG that depicts all the message chains in the design, whereas all the *black9* edges form another subgraph that characterizes all object-write flows in the design. We can study whether the small-world phenomenon [Kleinberg 2000], exists in the *black2* subgraph of the OO design. We can also investigate the link prediction [Liben-Nowell and Kleinberg 2003] and the phenomenon of dictating entities [Arrow 1963] in one or several subgraphs of the OO design. These issues (the small-world phenomenon, link prediction, and dictating entities) have been studied in various contexts including the social networks, world-wide web [Albert et al. 1999],[Kaiser 1999], [Kleinberg and Lawrence 2001], and generic graphs. It is reasonable to speculate that they may have some ramifications in the OO design network as modeled in the BBG framework.

References

- [1] R. Albert, H.Jeong and A.L.Barabasi, The diameter of the world wide web, *Nature*, Vol. 401, pp.130, 1999
- [2] Arrow K., Social choice and individual values, Cowles Commission Monograph 12, second edition, Vol.51, 1963
- [3] Davis J.A. Clustering and structural balance in graphs, *Human relations*, Vol. 20, pp. 180-187,1967
- [4] I. Jacobson and Pei-Wei Ng, Aspect-Oriented Software Development with Use Cases Addison Wesley, Upper Sanddle River, NJ, 2005.
- [5] J. Kaiser, It's a small web after all, *Science* Vol. 285, pp. 1815-1815,1999
- [6] J. Kleinberg, The small-world phenomenon: an algorithmic perspective, The Proceedings of the 32nd STOC 2000, pp. 163-170, 2000
- [7] J. Kleinberg and S. Lawrence, The structure of the web, *Science*, Vol. 294, pp. 1849-1850, 2001
- [8] W. Li, and H.Zhang, Towards a Formal Framework for Software Metrics, submitted to: *ACM Transactions on Software Engineering and Methodology*
- [9] D. Liben-Nowell, and J. Kleinberg, The link predication problem for social networks, The Proceedings of the Twelfth Annual ACM International Conference on Information and Knowledge Management (CIKM'03), pp.556-559, 2003
- [10] Management Group, Unified Modeling Language (UML), Version 2.0, <http://www.omg.org>, 2005.