Journal of Computer Science 4 (1): 59-65, 2008
ISSN 1549-3636
© 2008 Science Publications

# A Tool for Testing of Inheritance Related Bugs in Object Oriented Software

[1]B.G. Geetha, [2]V. Palanisamy, [1]K. Duraiswamy and [3]G. Singaravel
[1]Department of Computer Science Engineering,
K.S. Rangasamy College of Technology, Tiruchengode, India
[2]Government College of Technology, Coimbatore, India
[3]Department of Computer Science Engineering, KSR College of Engineering, Tiruchengode, India

**Abstract:** Object oriented software development different from traditional development products. In object oriented software polymorphism, inheritance, dynamic binding are the important features. An inheritance property is the main feature. The compilers usually detect the syntax oriented errors only. Some of the property errors may be located in the product. Data flow testing is an appropriate testing method for testing program futures. This test analysis structure of the software and gives the flow of property. This study is designed to detect the hidden errors with reference to the inheritance property. Inputs of the tool are set of classes and packages. Outputs of the tools are hierarchies of the classes, methods, attributes and a set of inheritance related bugs like naked access, spaghetti inheritance bugs are automatically detected by the tool. The tool is developed as three major modules. They are code analysis, knowledge base preparation and bugs analysis. The code analysis module is designed to parse extract details from the code. The knowledge base preparation module is designed to prepare the knowledge base about the program details. The bug's analysis module is designed to extract bugs related information from the database. It is a static testing. This study focused on Java programs.

**Key words:** Dataflow testing, inheritance property, class hierarchy, inheritance related bugs

## INTRODUCTION

**Software testing concepts:** Software Testing is the process used to help identify the correctness, completeness, security and quality of software. product against a specification. An important point is that software testing should be distinguished from the separate discipline of Software Quality Assurance (SQA), which encompasses all business process areas, not just testing. Whether software satisfies customers needs or not is a purpose of testing. Testing focus on two ways. (1) black box testing and (2) white box testing. white box testing exercise all independent paths, all logical conditions, execute all loops and exercise data structures. Black box testing are used to test that software functions are operational, that input is properly accepted and output is correctly produced. Control structure testing is a white box testing[1].

**Data flow testing:** One common approach to structural testing of software programs is to design and select test cases according to control flows of software programs. Common control-flow-based test coverage criteria include the statement coverage criterion, the branch coverage criterion and the path coverage criterion.

Data flow testing is a testing technique based on the observation that values associated with variables can effect program execution. Data flow testing not only explores program control flows but also pays attention to how a variable is defined and used at different places along control flows, which could lead to more efficient and targeted test suites than pure control-flow-based test suites. An important insight that data flow testing can provide is that it shows a way to distinguish between the useful ones and the less useful ones among test cases generated from pure control-flow-based testing techniques and trim the number of required test cases without reducing the effectiveness of the test suite.

**Inter procedural data flow testing:** Testing the data dependencies that exist among procedures (i.e., inter procedural) requires information about the flow of data across procedure boundaries, including both calls and

**Corresponding Author:** B.G. Geetha, Department of Computer Science and Engineering, K.S.R. College of Technology, Tiruchengode-637215, Nammakkal Dt, Tamilnadu, India
Tel: +91 98946 88866 Fax: +91 04288 2747451

returns. The data dependencies that exist between procedures both directly over single calls and returns and indirectly over multiple calls and returns are needed.

**Intra procedural data flow testing:** Testing within a procedure is called as Intra Procedure Analysis Testing[2].

## OBJECT ORIENTED DATA FLOW TESTING

Object oriented software is different from traditional software development. Object oriented development is a way to develop software by building self contained modules or objects that can be easily replaced, modified and reused. Each object had attributes and methods. Objects are grouped into classes. Basic concepts of object oriented programmed are data abstraction and encapsulation which is wrapping up of data and methods in to a single unit. Inheritance is the process by which objects of one Class acquire the properties of objects of another class. It supports the concepts of hierarchical classification. Polymorphism means ability to take more than on form.

**Class testing:** Class testing is the base of object-oriented software testing. It involves three aspects - testing each method, testing the relations among class methods and testing inheriting relation between class and subclass. In object-oriented programs, the methods are bounded (or encapsulated) within a large entity-class. So, testing each method independently is meaningless in object-oriented testing unless the relations among methods of a class and their joint effect on shared states are also tested. Hence, in object-oriented testing, the significant testing unit cannot be smaller than a class.

**Fragment class analysis:** The existing body of work on class analysis cannot be used directly to compute the RC and TM coverage requirements in a coverage tool. The key problem is that the vast majority of existing class analyses are designed as whole-program analyses-i.e., analyses that process complete programs. In contrast, testing is rarely done only on complete programs and many testing activities are performed on partial programs. Any realistic coverage tool should be able to work on partial programs and, therefore, needs analysis techniques beyond traditional whole-program class analyses.

To solve this problem, we need a class analysis that can operate on fragments of programs rather than on complete programs. We refer to such an analysis as a fragment class analysis. In previous research general method for constructing fragment class analyses for the purposes of testing of polymorphism in object oriented software using Java. Using this method, fragment class analyses can be derived from a wide variety of flow.

Insensitive whole-program class analysis. The significance of this technique is that it allows tool designers to adapt available technology for whole-program class analysis to be used in coverage tools for testing of polymorphism in partial programs[7,8].

**Static testing:** Static Testing is a form of software testing where the software isn't actually used. This is in contrast to dynamic testing. It is generally not detailed testing, but checks mainly for the sanity of the code, algorithm, or document. It is primarily syntax checking of the code or and manually reading of the code or document to find errors. This type of testing can be used by the developer who wrote the code, in isolation. Code reviews, inspections and walkthroughs are also used.

**Static code analysis:** Static code analysis is the analysis of computer software that is performed without actually executing programs built from that software (analysis performed on executing programs is known as dynamic analysis). In most cases the analysis is performed on some version of the source code and in the other cases some form of the object code. The term is usually applied to the analysis performed by an automated tool, with human analysis being called program understanding or program comprehension.

The sophistication of the analysis performed by tools varies from those that only consider the behavior of individual statements and declarations, to those that include the complete source code of a program in their analysis. Uses of the information obtained from the analysis vary from highlighting possible coding errors (e.g., the lint tool) to formal methods that mathematically prove properties about a given program (e.g., its behavior matches that of its specification).

Some people consider software metrics and reverse engineering to be forms of static analysis. A growing commercial use of static analysis is in the verification of properties of software used in safety-critical computer systems and locating potentially vulnerable code.

## THE INHERITANCE CONCEPT IN OBJECT ORIENTED SOFTWARE

In object oriented programming, objects will be characterized by classes. It is possible to learn a lot about an object based on the class it belongs to. Objected oriented programming takes this concept to a Whole new level .It permits classes to be defined in relation to other classes. Every subclass will inherit a state from the super class. Despite this, subclasses are not restricted to the behaviors and states that they have taken from their super class. A subclass can combine methods and variables with the traits they have inherited from their super class. It is also possible for subclasses to override any methods that they have inherited, and they can create unique implementations for these methods. It is also possible to use more than just one level of inheritance. An inheritance structure can be generated which can be as deep as you want it to be. This inheritance structure is called a class hierarchy. The variables and methods can extend through the levels of the class hierarchy. In most cases, a hierarchy that is deep tends to have behaviors which are distinct. It should always define what the classes are instead of how they are used. The object class should be at the zenith of the class hierarchy. Every class should descend from it in a direct or indirect manner. The variable of an object type can retain a reference for any object, and an example of this would be a class. For example, the object could define behaviors that may be attributed to the objects that are processed by the Java Virtual Machine. There are a number of power advantages to the concept of inheritance. Subclasses can generate distinct behaviors which are based on the common attributes that are present in their super class. Because of inheritance, it is possible for programmers to use the same code many times over. Programmers can generate super classes which are named abstract classes. Abstract classes will characterize behaviors which are common. While some aspects of this behavior may be defined, a large portion of it will not be defined at all. It shows how subclasses are connecting to their super classes, and it can also allow you to understand which traits have been passed from the super class to its subclasses. It is one of the most powerful features of object oriented programming. It is used in a number of popular programming languages such as C++, Java, Small talk, Objective-C. It is features like this that makes OOP a powerful tool that many programmers use to create important programs. However, it is just one of the few concepts that you must understand if you wish to use this programming paradigm [3].

**Types of Inheritance: Inheritance is also sometimes called generalization, because the is-a relationships represent a hierarchy between classes of objects**

**Single Inheritance**

- Derived class has only one direct base class
- Creates "simple" hierarchy of classes - trees
- One to one inheritance of members
- Specializes a base class

**Multiple Inheritance**

- Derived class has more than one direct base class
- Creates "complex" hierarchy of classes - graphs
- Possible multiple inheritance of members
- Combines multiple classes
- Same Inheritance and Access Rules
    - ➢ Derived class contains all members from all base classes
    - ➢ Regardless of access modes

**Inheritance Conflicts :**

- Member Conflicts
    - ➢ Name conflicts can occur - same member name from more than one base class
    - ➢ Derived class can overshadow base class members name
    - ➢ Use scope resolution operator to resolve conflicts
- Multiple Inheritance Conflicts
    - ➢ Derived class may combine more than one copy of a member
    - ➢ Base class may combine more than one copy of a member

**C++ and Java difference in terms of Inheritance**

| **C++** | **JAVA** |
|---|---|
| C++ supports multiple inheritance of arbitrary classes | In Java a class can derive from only one class, but a class can implement multiple interfaces |
| In C++ multiple inheritance and pure virtual functions makes it possible to define classes that function just as Java interfaces do. | Java explicitly distinguishes between interfaces and classes |

## INHERITANCE RELATED BUGS

**The following are some common inheritance related bugs.**

Incorrect Initialization: Super class initialization is omitted or incorrect Deep hierarchies may lead to initialization bugs. Determining how initialize is used in a subclass requires examination of the super class that defines new. The initialize message must be sent to super, not self. Suppose that new is refined and does not send initialize to it. Super's initialize will not be executed.

Example

```
Class shape
 {
 Public
 Virtual void draw();
 };
Class Rect: public shape
 {
 Public:
 Void draw();
 …….
 };

Void main()
{
 Shape s; /* Incorrect initialization*/
 …….
}
```

In the above example, the object for 'shape' class is created in 'main' function. It doesn't accept for drawing an object. The 'shape' is an abstract or base class. So it provides only information about the shape object.

**Inadvertent bindings:** Incorrect bindings can result from misunderstood name scooping rules the bindings of names under multiple inheritances introduces more subtleties.

**Missing override:** A subclass specific implementation of a super class method is omitted. As a result, that super class method might be incorrectly bound to a subclass object and a state could result that was valid for the super class but invalid for the subclass owing to a stronger subclass invariant.

Example

```
Class shape
{public:
Void area(); /* missing virtual keyword*/
};
Class circle:public shape
{
…….
Public:
Void area();
……
};
```

If base and derived classes are having member functions in same name may make function overriding. So avoid function overriding, use 'virtual' functions to execute both base and derived class member functions.

**Naked access:** A super class instance variable is visible in a subclass and subclass methods update these variables directly. Naked access creates the same problems as unrestricted access to global data. Changes to the super class implementation can easily induce subclass bugs or side effects. Subclass bugs or side effects, in turn, can cause failures in super class methods.

**Square peg in a round hole:** A subclass is incorrectly located in a hierarchy.

**Naughty children:** A subclass either does not accept all messages that the superclass accepts or leaves the object in a state that is illegal in the superclass.

**Worm holes:** A sub class values that are not consistent with the super class invariant or superclass state invariants. The state space of lower classes of a well formed class hierarchy must not expand on superclass state space.

**Spaghetti inheritance:** A number of multiple inheritance and very deep hierarchies (More than 5 Levels) are error prone, even when they conform to good design practice. The wrong variable type, variable, or method may be inherited[2].

## TOOL DESIGN

The inheritance bug identification tool is developed as a graphical user interface based system. The system is designed to analyze the Java based source code only. This analysis is called as static analysis. The system implementation is carried out by using the Java language and Microsoft Access back end tool. The system is designed to analyze any third party and Sun Microsystems open source code. The knowledge model is updated for each test cycle. Future test cycles use the knowledge model details.

Three major modules are used in the system implementation. They are the code analysis, knowledge model preparation and the bug analysis. The code analysis module is developed to extract details from the source code files. The knowledge base preparation module is developed to update the knowledge base in an organized manner. The bug analysis is done on the source code details that are maintained in the database. The system uses the product path as the input. The system products a list of bugs with its occurrence details.

**Code analysis:** The code analysis is the initial module for the system. The product path is given as the input for the system. The Java source code files are identified first. Then each file content is fetched from the file. The noise filtering is performed after the code fetching process. The documentation comments and general comments from the source are removed. These comments are called as noise in the source code. The filtered code and class details are extracted. The package details and class details are updated into the database.

**Knowledge base preparation:** The knowledge base is a collection of source code elements for all Java programs in the product source code. The packages are the top level elements in the knowledge base. Each class details like method and attribute details are collected and updated into the database. The class relationship with other classes are also maintained separately. The interface for the class details are also collected and maintained in the database. The attribute details include the name of the attribute, type of the attribute, modifier details. The method details also include the method name, argument details and return type values.

**Bug analysis:** The bug analysis module is designed to detect the hidden errors in the source code. Incorrect Initialization, inadvertent bindings, missing override, naked access, naughty children and spaghetti inheritance and Fat Interface bugs are detected by the system. The bugs are related to the inheritance concepts. Each type of bug is detected for the source code and listed in a separate form.

Already number of tools are available for testing of object oriented software's. In this study developing a data flow testing tool for testing of Inheritance property. Input of the tool is set of procedures, set of classes or packages. Output of the tool is class list, attributes and methods list for particular class. It shows the hierarchy of the classes.

**Experimental results:** For convenient Java software is taken for testing. Tool is developed as three modules.
1. Code analysis 2. knowledge base preparation 3. Bug analysis.

In code analysis tool read every token and store it in a database. After reading documents lines are eliminated and store it in separate database. Tool search for key words related to class, methods, attributes and inheritance declaration. All are stored in data base. Knowledge base contains information about document,

Eliminated source code, keywords, size of the class, lines of code in each class. Finding relation first one class to same class, one class to next level class checking for inheritance relationship up to 'n' level. In java multilevel inheritance available. Instead of multiple level interfaces are used. In this focus towards multilevel inheritance. The output of the relation displayed as a matrix which contains Boolean value. Number of rows and columns equal to number of class hierarchies.

For Experiment Standard Sunsoft Java 1.4 Software have taken and experimented.

## MAIN SCREEN OF THE TOOL

Table 1: Samples java files with size after removal of document lines

| Size | |
|---|---|
| Applet Java | 17641 |
| Applet Context Java | 6887 |
| Applet Stub Java | 2780 |
| Audio Clip Java | 866 |

Table 2: Number of Classes in Each Package

| Package Name<br>Classes | |
|---|---|
| Applet | 4 |
| Callback | 10 |
| CORBA | 9 |
| Event | 44 |
| Jar | 8 |
| Reflect | 57 |
| Zip | 18 |

Table 3: Number of lines in class after removal of document line for analysis

| ATTRIBUTE DESCRIPTION |
|---|
| Private long stem; |
| Private int off, len; |
| Private int level, strategy |
| Private boolean setParams; |
| Private Boolean finish, finished |
| Public static final int DEELALTED = 8; |

Table 4: Attributes in classes

| | | |
|---|---|---|
| Class Name | | |
| File Size | | |
| Line of Code | | |
| Action Event | 7237 | 220 |
| Adjustment Event | 6430 | 232 |
| AWT Event Listnernpr | 1745 | 62 |
| Component Adapter | 2049 | 56 |
| Componentevent | 4747 | 140 |

Table 5: Methods in class

| METHOD DESCRIPTION |
|---|
| Public Deflate (intlevel, Boolean nowrap) |
| Public Deflate (int level) |
| Public Deflater( ) |
| Public synchronized void setinput (byte[]b,iny ogg, iny lrn) |
| Public void setinput(byte[]b_ |
| Public synchronized void setDictonary[]b, int off, intlen) |

Table 6: Classes with four level

| | |
|---|---|
| Action Event | |
| Inheritance Hierarchy | |
| Java.awt.event.ActionEvent | 4 |
| Java.awt.AWTEvent | 3 |
| Java.Util.EventObject | 2 |
| Java.lang.Object | 1 |

Table 7: Classes with six level

| | |
|---|---|
| KeyEvent | |
| Inheritance Hierarchy | |
| Java.awt.event.KeyEvent | 6 |
| Java.awt.Event.inputevent | 5 |
| Java.awt.event.componentEvent | 4 |
| Java.awt.AWTEvent | 3 |
| Java.Util.EventObject | 2 |
| Java.lang.Obejct | 1 |

Table 8: Classes with Seven Level

| | |
|---|---|
| MouseWheelEvent | |
| Inheritance Hierarchy | |
| Java.awt.event.mousewheelEvent | 7 |
| Java.awt.event.mouseEvent | 6 |
| Java.awt.event.inputEvent | 5 |
| Java.awt.event.ComponentEvent | 4 |
| Java.awt.AWTEvent | 3 |
| Java.Util.Event.Object | 2 |
| Java.lang.Object | 1 |

From the Table 7 & 8 the level increased more than five. So there is a possibility to Spaghetti Inheritance error. Table 6 can be Represented in Matrix Format for inheritance relationship.

$$\begin{pmatrix} 0 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

## CONCLUSION

Object oriented programming system is the popular software development mechanism in the recent days. Inheritance is one of the important features for the object oriented systems. A class can be inherited by another class. Multiple level and multilevel of inheritance are used in a product. In this case there is a chance to errors in the product. The compiler does not detect property errors. The compiler only detects the syntax errors.

The system is developed as an automation tool for the static testing process to the Java language. The open source code for the Sun Micro system is tested using the system. All the code details are updated into the knowledge base. The system is also tested with some other third party software products. The system detects a hidden inheritance related bugs.

The compiler checks the syntax errors and converts the source code into byte code. But the compiler is not a complete solution for the error detection requirements. This system is developed to test the Java based products as a static testing tool. The static test is applied to detect the inheritance related bugs in the Java programs. The system can be enhanced with the following features.

- The current system is designed to find out the inheritance related bugs from the Java products only. In future the system can be enhanced to detect multiple inheritance hierarchy.
- Tool can be developed for other object oriented softwares like Smart Talk, Objective -C.
- The current hidden bug detection scheme can be integrated to a compiler to detect the hidden errors during the compile time.
- The knowledge base model can be used to find object oriented metric analysis for quality product.
- The current system is developed as static analysis tool to test the source code. The same concept can be implemented under the dynamic testing mechanism to analyze the product using the byte code to analyze the third party products.

## REFERENCES

1. Binder, R. Testing object oriented software: Surve, software testing, verification reliablity, 6 125-252.

2. Binder, 1999. Testing Object-iented Systems: Models, Patterns and Tools, Addison Wesly.
3. Chun-Chia Wang and Wen C. Pai. An Automated Object-Oriented Testing for C++ Inheritance Hierarchy. Department of Information M Management Kuang Wu Institute of Technology and Commerce.
4. Harrold, M. J.D. McGregor and K. Fitzpatrick, 1992. Incremental testing of object-oriented class structure. In: Proceedings of the 14th International Conference on Software Engineering, pp: 68-80.
5. Jacobson, I., M. Christerson, P. Jonsson and G. Overgaard, 1992. Object oriented Software Engineering: A Use Case Driven Approach, Addison-Wesley, Reading, MA.
6. Perry, D.E. and E.G. Kaiser, 1990. Adequate testing and object-oriented programming, J. Object-Oriented Programming, 2 (5): 13-19.
7. Rountev, A., 2002. Dataflow Analysis of Software Fragments, Ph.D Thesis Rutgers Univ., Aug 2002.
8. Rountev, A. Milanova and B.G. Ryder. Fragment Class Analysis for Testing of Polymorphism in Java Software.
9. Smith, M.D. and D.J. Robson, 1992. A framework for testing object-oriented programs, J. Object-Oriented Programming, 5 (3): 45-63.