The Experimental Evaluation of Interclass Testing for Object Oriented Programming

Prof. Marian CRISTESCU, PhD
The "Lucian Blaga" University of Sibiu

The paradigm of object oriented is successfully applied in many software projects and the use of the languages object oriented widely spread nowadays. The object oriented technologies diminish or eliminate some typical problems of the procedural software, but on the other hand they can introduce new problems wich have as a result classes of defects difficult to be adressed by means of traditional technologies of testing. In particular, the defects depended of the state, are to be meet more frequently in the object oriented programming systems rather than the procedural ones; almost all object have an associated state, and the behavior of the object. Such defects can hardly be decovered because they create anomalies only when the objects are used in particular states.
***Keywords**: object oriented technology, interclass testing, reliability, flow graph analysis.*

Introduction

A relation of assembly between two classes A and B appears when a A type object can include one or more B type objects. In this case the state of an A type object depends on the state of the B type objects witch it contains.
A relation of utilization between two classes A and B intervenes when one or more A type methods have at least local variable or a parameter of B type.
The technique presupposes the generation of some appeal sequences for a set of objects which form a subsystem. The generated sequences cover pairs of methods which modify and utilize the state of some object. Due to this thing, the sequences exercise the object in subsystem in different states and they can discover errors which appear only when the class objects find it in particular states.

Interclass-testing

The interclass testing is the testing of a set of classes which form a system or a subsystem usually procesed during the phase of integration. Such classes are usually independent entities but they mutually cooperate in different ways.
These relations harmony classes represent a fundamental characteristic of the oriented objects sysytems during execution. These are various classifications of interclass relations and one of the most interesting relations is the relation of assembly and utilization.
In order to extend this technique of generating interclass tests to the interclass case is to consider the mentioned relations between objects: the relation of assembly and utilization. The following aspects are to be extended in particular:

- The concept of associating the definition and utilization in the case the variables which are not scalar entities but objects.
- The flux analysis of interclass dates in order to analyze a set of classes incrementally by reutilizing the total information per each class. At the beginning the technique produces specifications for experimental cases automatically and later it generates possible cases of testing for the produced specifications.

The generation of specifications for testing cases

The specifications of experimental cases for interclass testing, generated by means of the former methods are described as pair methods in which the former method modifies the state of the object while the second method accesses the modified state. The pair methods are generated by:

- The identification of some classifying which allows the incremental analysis of the data flux;
- The performing of some incremental analyses of data flux to permit the work with classes whose state comprises instances of other classes.

Classification of classes - in order to perform the incremental analysis of the data flux at interclass level, a class is to be analyzed at a certain moment, and the result of the analyzed of each class is to be totaled. Then the classes which are utilized or are included in the other classes must be analyzed before the classes which they use and certain. That is why the first step taken in the interclass analysis, is the identification of an order by sequence of the analyses leased on the relation of assembly and utilization between two classes.

The two binary relations over the set of C classes define an oriented graph whose knotes are the classes of the set and whose edges represent the relation of utilization and/or assembly. Then it is presupposed that the graph is conex. A noncom graph would imply the presence of independent subsystem which could be analysed separately.

Within the systems of object oriented programming, well-designed, the structure and the dependences should have as a result a **DAG** (Data Analysis Graph) where there is a partial order of the elements and a total typological order aver such elements is possible. The analysis of the classes in concordance with this total order permits the analyses of utilized class before the classes which use it and an included class before the classes which contain it. If the graph contains cycles they can be eliminated by deleting one or more edges and by manual supply of the information which is calculated from the vertex automatically.

The interclass data flux analysis – an association between definition and utilization is a triplet (*d,u,v*) where *d* and *u* are declarations and *v* is a variable, *d* defines *v*, *u* utilizes *v*, and there is a way from *d* to *u* where *v* is not redefined. The techniques of traditional data flux, used for procedural programming systems were extended by Harold and Rothermel in order to manipulate sequences of object oriented code [HARR94]. In order, to realize the interclass testing, this technique is applied with a view to calculating a subset of association's definition-utilization for a class. The subset which presents interest in calculating contains all associations of definition-association which involve variables of scalar instance of the class.

Example 1:

```
class Foo{
 public:
        int x;
1.      void incX(){
2.       x++;
        }
3.      int getX(){
4.       return x;
        }
};
```

```
class Bar{
private:
        Foo foo;
    public:
    void m(){
    5.      foo.x=0;
    6.      foo.incX();
    7.      print(foo.getX());
            }
    };
```

A definition of *foo* object a line 5 can be easily identified in this example but the effect of line 6 cannot be appreciated. The syntax of line 6 is possible to contain a definition of *foo*, an utilization of *foo*, or both of them, this depending on the semantics of method *Foo::incX()*. That is why the analyses of line 6 require initial analyses of *Foo* class. The some consideration is maintained of line 7.

On the other hand after analyzing *Foo* class, a classifications of method can be made, dividing them into methods which define, utilize or define and utilize the attributes of *Foo* (for example: method which modify, inspect or modify and inspect the state of *Foo*).

For this example, the method *Foo::incX()* is classified being a method which inspect and modifies the state of *Foo* and *Foo::getX()* as

a method which inspects the state of *Foo*. Using such cumulative information, the utilization and the definition of *foo* an line 6 and the utilization of *foo* an line 7 can be identified correctly.

Generally speaking, it can be asserted that an object is defined, respectively utilized in a declaration when any of its member variables is defined and utilized in the declaration.

Example 2:

```
class Complex;
class Foo{
 public:
        complex x;
        void incX(){
                x. incRe();
        }

};
```

```
class Bar{
        private:
        Foo foo;
        public:
         void m(){
                …
8.      foo.incX(); …
                }
};
```

In order to have the possibility of analyzing *Bar* class scant information about class *Foo* (direct dependence) and class complex (indirect dependence through class *Foo*) is necessary. Therefore, the calculation of the concise information for a certain class requires the classification of class methods as being: modifier, inspector or inspector and modifier at the same time, in concordance with the effect the implementation of the method has on the state of the class.

A method as a modifying role if its appeal causes a modification of the class state, The latter determining a change of the value of a member class variable or an appeal to a modifying method of a member variable. One method is known as inspector if its appeal determines the utilization of the value of a member variable or he appeal of an inspecting method of a member variable. A method can be considered inspector-modifier if it is both: an inspecting method and a modifying method. The methods independent of the state of the class (for example: the methods which neither modify nor use instances variables) are ignored during the incremented analyses. So, in order to analyze a C class, there appears his necessity of calculating some scant information referring both to the completion of assembly and utilization relations which start from C class. The find ordering of all classes in the systems allows the efficient performing of some calculations, by permanently analysis a class before other classes which depend on it.

This definition is recurrent because a member variable can be reutilized in an object.

Therefore, before analyzing C class, concise information for *S* set of the classes it depends on, directly or indirectly is necessary. The following version of classes *Foo* and *Bar* are taken into consideration:

In order to determine the various characteristic of his ways a method may offer you, each way of a method can be considered as if it were different method of the class which applies the data flux analysis to each way. This is a beneficial hypothesis to small and average programming systems, but it is unworkable with large and complex applications because all the combinations of the ways identified as formal steps are to be considered. A similar problem arises when using the traditional techniques of data flux testing: the member of combinations of definitions and utilization can increase in the system of great programmers.

The generation of case test

The definition-utilization associations, created as a result of a data flux analysis, identify a set of testing cases necessary to the performing of an interclass testing. A testing-case, corresponding to his definition-utilization association is a sequence of appeal of a method which begins with a constructor and includes the appeal of both methods arisen as a result of a definition-utilization association.

The symbolic interclass execution – as a result of the previous phase a set of pairs of methods which define and use the states of the given objects will be provided. For each

pair, the first method contains a definition of the object's state whose it belongs to and which is known as *definition*. In the same way the second method contains a utilization of the object's state which can be either direct or indirect, through an appeal of an inspector method and which can be also referred to as *utilization*. The symbolic execution determines the conditions which lead to the executions of the different ways inside each method; attention is given to the ways which contain the definition and utilization identified to the previous phase. It is applied to every method of each class by using the principle of – a method in turn. The symbolic execution allows the calculations of the following elements for every way of each method: the condition associated with the executions way and the relations between the *in* and *out* values of the method observing the chosen way.

Generating the testing cases – is the last phase of the described technique in [KCMC00] and [JOER94] and it consists in generating sequences of appeal of the method which practice each defining-utilizing association. In order to be very clear the following notations are used:

- u - declaration which contains an utilization of variable v;
- d - declaration which contains a definition of variable v;
- m_u – method which contains declarations u;
- m_d – method which contains declarations d;
- PCU – the condition of executing the way of each m_u appears;
- the method without definitions, which accounts for variable *v* – it is a method which does not determine a redefining a variable *v*;
- the way without definitions, which accounts for variable *v* – it is a sequence of methods without definitions which take into account variable v.

Taking into consideration the defining and utilizing association (*d,u,v*) for a C class, a sequence of appeals of the methods which performs the association must satisfy the following properties [JOER94].

- to begin the appeal of the constructor of C class;
- to contain an appeal of method m_d which causes the execution of declaration *d*;
- to contain of an appeal of method m_u which causes the executions of declaration *u*;
- the sequence of appeals between m_d and m_u must be a well-defined way which observes the declaration of variable v.

Corresponding to [KCMC00] it is to follow the following actions in order to calculate the new objective:

- the simplification of the current conditions by eliminated those classes which are satisfied by post conditions m_u;
- the reunion of the simple conditions and preconditions m_k;
- the simplifications of the resulted conditions where possible.

Because a series of adequate methods m_k can exist in each phase, the deductive process devoted to achieving a defining-utilizing association can be represented by a tree-like structure. Each knot corresponds to a pair made up of a method and a condition (a predicate applied to his instance variables of a class and to the parameters of a method). The root of the tree cares ponds to the m_u method and to the PCU condition. The first objective of the deductive process is to add as knots which correspond to the m_d method of the tree. The second objective is to add a constructor to he sub tree root of m_d method. The deductive method is over when the second objective is fulfilled. A restriction to the deductive process is that only the methods without definitions and those which take into consideration variable *v*, can be added to the tree before fulfilling the first objective (for example: the way between m_d and m_u must be well-defined).

The deductive process can be accomplished due to the following 3 reasons:

- both objectives are satisfied, which means that the process of searching a freezable sequence of methods has been accomplished successfully;
- the tree cannot be extended any longer, and this means that the defining-utilizing association will not be freezable;

• the depth of the tree reaches the extreme limit before a fezabil sequence is to be found.

In order to improve the efficiency of the deductive process the use of the heuristic methods is recommended. First of all, the deductive tree's dimension is reduced by cleaning the sub trees whose roots have conditions which imply preconditions. This desideratum is achieved by avoiding the further exploration of this kinds of sub trees where they correspond to other methods m_d or other constructors and their inclusions in the tree represents top priority objective of the deductive process.

Secondly, the insertion of a knot corresponding to the responsible method for defining (for example m_{ad}) is advised as soon as possible the unique successor.

Conclusions

The technique of building the tree described in this paper is based on the use of the automatic deduction. In situation of the existence of an automatic technique of deduction, such a solution subjected to constraints, could fail during the operation of copying which use complex expressions. At present, there are many efficient ways of eliminating the constraints which allow the efficient maneuvering of some extended sets of expressions and this makes possible for the appear failure to be overloaded due to the intervening necessities on behold of the utilizes.

References

[HARR94] - Harrold M.J. and Rothermel G., "*Performing data flow testing on classes*", In 2nd ACM-SIGSOFT Symposium on the foundations of software engineering, ACM-SIGSOFT, December 1994, pp.154–163;

[JOER94] - Jorgensen P. and Erickson C., "*Object-oriented integration testing*", Communications of the ACM, 37(9):30–38, September 1994;

[KCMC00] - Kim S., Clark J.A., and McDermid J. A., "*Class mutation: mutation testing for object-oriented programs*". In Proceedings of the NetObjectDays - Conference on Object-Oriented Software Systems, 2000;

[SPHI99] - Souter A., Pollock L., and Hisley D., "*Inter-class Def-Use analysis with partial class representations*", In Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, vol. 24 of Software Engineering Notes *(SEN*), September 1999, pp. 47–56.