International Journal of Networked and Distributed Computing, Vol. 3, No. 1 (January 2015), 11-20

The Influence of Alias and References Escape on Java Program Analysis

Shengbo Chen [1, 2], Dashen Sun [1, 2], Huaikou Miao [1, 2]

[1] *School of Computer Engineering and Science, Shanghai University, Shanghai 200436, P. R. China*
[2] *Shanghai Key Laboratory of Computer Software Testing and Evaluating, Shanghai 201112, P. R. China*
E-mail: {schen, sundashen, hkmiao}@shu.edu.cn

Abstract

The alias and references escape are often used in Java programs. They bring much convenience to the developers, but, at the same time, they also give adverse affects on the data flow and control flow of program during program analysis. Therefore, when analyzing Java programs, we must take the alias and references escape into account. This paper proposes a static approach to detecting control flow information of programs with alias and references escape. Firstly, it computes the data flow information, including def-use information and alias information caused by references assign and references escape. Secondly, it analyzes the program and gets the control flow information based on the obtained data flow information. Finally, a case study is presented to show that the proposed method can detect control flow information exactly.

Keywords: reference escape, program analysis, control flow, data flow, alias information.

1. Introduction

Different programming languages have their own special features. such as variable scope, exception-handling constructs, pointer alias and references escape. These features give a convenient way for the developers to implement the functions of the program. Java is a popular programming language that integrates many useful features. specially, the alias and references escape are wildly used in Java programs when we develop software applications using Java programming language. But usually they may change the control flow of the corresponding programs. As a result, the data flow and control flow information of the program which we get may be incorrect if we do not take the alias and references escape into account when we analyze the Java programs.

Program analysis is the process of analyzing the behavior of a computer program. Program analysis has a very widely application range, it provides support for compiler optimization, testing, debugging, verification and many other activities. There are many program analysis techniques, but according to the principle "whether is needed to run the program or not", these techniques could be divided into static analysis techniques and dynamic analysis techniques. Static program analysis is performed without actually executing the existing programs. Generally, the static analysis object is source code.

Published by Atlantis Press

Dynamic program analysis is performed by the way of executing the programs, and the target programs must be executed with sufficient test inputs to produce expected outputs. Because of dynamic analysis needs to execute the programs manually, it needs too many test cases and costs too much. Especially for large programs, the disadvantages of dynamic analysis are more obvious. Current analysis techniques are mostly based on static analysis, and in this paper we focus on static program analysis.

Data flow analysis and control flow analysis are the two most widely used static analysis techniques for program analysis. Both of the two methods refer to data flow information and control flow information. Therefore, current popular analysis techniques often combines these two methods. Data flow analysis is used to accurately describe the functions of system, input, output and data storage logically. Usually, data flow diagram (DFD) is employed in data flow analysis. DFD graphically describe the flow of data and data process which is an important part of the system logic model. While control flow analysis is a static code analysis technique for determining the control flow of a program. As usually, the control flow is expressed as a control flow graph (CFG) [1,2]. In this paper, we use data flow analysis to get data flow information, and then we use the data flow information to get control flow information. For Java programs, if the influence of alias and references escape is ignored, the data flow information and control flow information we get may be influenced. Therefore, in order to improve the accuracy of data flow analysis, and get the control flow information of the program, we must take the influence of alias information and references escape into account.

In this paper, we propose an approach to getting control flow information of Java programs using static programming analysis with large numbers of conditional branch statements and alias and reference escape are been considered. The method is based on traditional data flow analysis technology and it takes alias information and references escape into account. It first constructs the control flow graph of the program by identifying the basic blocks of the program, then it computes the `def-use` information and alias information, and finally it uses the `def-use` information and alias information to analyze the program and get the control flow information. From the results of the program analysis, all of the feasible and infeasible program paths are given out.

This paper is organized as follows. Section 2 gives some primary knowledge about alias and reference escape which will be used in the rest of this paper. The def-use information and alias information are presented in Section 3. The case study and analysis of our approach are given out in Section 4 which shows the influence of alias and references escape on Java program analysis. And Section 5 states some related work. Additionally, some conclusion remarks and future work are given out in Section 6.

2. Primary Knowledge

In Java programs, there are two data types: primitive types and reference types. Primitive types define a range of basic data values that can be stored in a variable. The reference types define references to objects of classes, which contain collections of variables and methods that are described by the classes. Data of primitive types can be of either arithmetic or Boolean type. Class and array types are the examples of reference types. Any declared variable in a Java program can be of any of the primitive or reference types. In the rest of the paper, we refer to the reference types as usage types.

When reference types are used in a program, alias or references escape may occur. And the data flow information and control flow information of the program may be influenced. So, we will study how the alias and reference escape influent the program paths. For the sake of facilitating understanding the proposed approach, the preliminary knowledge is given out in this section.

2.1. Reference Escape

If an object is created in a corresponding method, and immediately it is assigned to a non-local variable or a field of non-local variable, or it is passed to another method as the form of parameters or passed out of the internal method as return values, then the

lifetime of the object is beyond to the lifetime of its creation environment. The objects which meet the above conditions are called *escaped objects*.

In this paper, we focus on two different kinds of escape information: 1) A reference escapes if it is returned to other part of the program. 2) A reference escapes if it is passed as a parameter to a method [3]. If an object is created outside the current scope and is accessed via a reference created outside the current scope, the object is already accessible to some part of the current scope. In this case we say that the object has escaped. If an object has not escaped but will be returned via a reference by the method to its caller, we say that the object will escape. Here, we give two classes *A* and *B* as the examples to address our method, as shown in Fig. 1.

```
Class A{
      Public void change(Student a){ … }
      Public void something( ){
      Student s = new Student();
      this.change(s);
       …
       }
}
```

(a). Example one.

```
Class B{
      Student doSomething() {
      Student s = new Student();
      return s;
      }
}
```

(b). Example two.

Fig. 1. Motivating Examples of Reference Escape.

In Class *A*, a method *something*() is located in it, and in the method, an object is created and we use a reference *s* point to it. Then the reference *s* is passed into another method *change*() as a parameter in the same class. Then the method *change*() can use the reference *s* to change the state of the object which *s* points to, but other regions know nothing about this operation except the method *change*(). Therefore, we say that the reference *s* is escaped into the method.

In Class *B*, a method *doSomething*() is located in it. An object is created within the method and it has a reference *s* pointing to it. Different from the first example, it then returns *s* to the method as a return value. For the caller of this method, it can get the reference *s* and change the state of the object which *s* points to. Therefore, we say that the reference *s* will escape out of the definition method as a return value.

2.2. Aliases

In this paper we focus on two different types of aliases in Java program. One type of aliases occurs when a variable with reference types is assigned to another variable, the two references share the same memory location, such as statement "$p = q$". The other is implicit aliases caused by references escape in method calls. In the previous section, we point out that there are two types of references escape: An object escapes if it is returned to other part of the program or passed as a parameter to a method. In this part, we regard both of the two types of references escape as a kind of reference aliases. For example, if a reference p is passed as parameter q to a method, then q is regarded as an alias of p. Similarly, in statement "$q = a.change()$", if the method *change*() returned a reference p to its caller q, then p and q actually alias.

For both of the two types of aliases, the reference p points to the object which the reference q points to, and here we can give the definition of aliases in Java programs.

Definition 1. [aliases] Two or more references are bound to the same object in some executions of the program.

When aliases occur in a program, and one reference changes the state of the object it holds, the other object holders may be affected by this change in the condition that they do not know what have happened, the results which caused by this operation may be fatal.

3. Data Flow Information

In order to get the control flow information of the program, first, we construct the control flow graph,

and then we use data flow analysis to analyze the data flow of the program, finally we will get the control flow information. At present, there are many programs which have large numbers of branches, and each branch can determine a control flow. Which branch to choose is determined by the value of the conditional statement, and the value of the conditional statement refers to the data flow information. Therefore, in order to get control flow information, we should firstly get the data flow information of the program.

Most approaches of data flow analysis involve decomposing the whole-program analysis into several sub-analyses of individual components, for example blocks, then summarizing the results of these sub-analyses, and memorizing those results for possible later re-use in other calling contexts. Basic block is a basic unit during program execution [4], each basic block has only one entrance and one exit statement, and could only contain no more than one conditional branch statement. The first step of our approach is to compute the `def-use` information and alias information of each basic block in the control flow graph.

3.1. Def-use Information

Data flow analysis is a common technique for statically analyzing programs and it is used widely for a long time. Traditional data flow analysis include def-use information: 1) reaching definitions, which propagates sets of variable definitions that reach a program point without any intervening writes, and 2) liveness, which determines the set of variables at each program point that have been previously defined and may be used in the future [5].

In Java programs, if the data type of the variable is primitive types, its value could be changed when it is used. But if the data type of the variable is reference types, when it is used, the state of the object which the reference points to may be changed. Here are some sets [6] used in this paper:

- $Def[B]$: the set of variables which is defined or assigned in block B;
- $In[B]$: the set of definition variables which could reach block B;
- $Gen[B]$: the set of variables which is defined in block B and could reach the exit of the block;
- $Kill[B]$: the set of variables which has been defined previously and defined again in block B;
- $Out[B]$: the set of variables which can reach the exit of block B.
- $Use[B]$: the set of variables which is used in block B.
- $Pred[B]$: the set of blocks that immediately proceed B in the control flow graph.

Each block has an associated in and out data flow set, and some other sets which has been listed. The sets including $Def[B]$, $Gen[B]$, $Kill[B]$, $Use[B]$ and $Pred[B]$, are easy to compute according to the basic information of block B. $In[B]$ is the data flow set of definition variables that reach block B, it is related to the set $Pred[B]$. $Out[B]$ is the set of variables which can reach the exit of block B, it has something to do with set $Gen[B]$, $In[B]$ and $Kill[B]$. Both of $In[B]$ and $Out[B]$ can be received by iteratively evaluating the equations until convergence to a solution. The equations [7] are shown as follows:

$$In[B] = \bigcup_{p \in pred[B]} Out[B] \tag{1}$$

$$Out[B] = Gen[B] \bigcup (In[B] - kill[B]) \tag{2}$$

3.2. Alias Information

In Java programs, due to the usage of references type variables, there may be a large number of alias information. In section 2, we mentioned that there are two types alias information: one is caused by assigning a reference to another reference, the other is implicit aliases information caused by references escape in method calls. In this section, all alias relations are of the form $\langle p,t \rangle$, where p and t are references that represent the same object. In most intraprocedural alias analysis, the following sets [8] are used to compute the alias information:

- $Pred(B)$: the block that immediately proceed B in the control flow graph of the program.
- $GEN\_Alias(B)$: the set of aliases that generated in block B and can reach the exit of the block.
- $KILL\_Alias(B)$: the set of aliases that destroyed in block B.
- $IN\_Alias(B)$: the set of aliases that can reach the entry of block B.
- $OUT\_Aias(B)$: the set of aliases that can reach the exit of block B.

Here are some explanations about the alias information set:

$$IN\_Alias(B) = OUT\_Alias(p), p \in Pred(B)$$
$$GEN\_Alias(B) = \{\langle p,t\rangle | p \in Alias(p), t \in Alias(t)\}$$

Our method first analyzes all methods which may be called later and summary the alias information at the exit of the method. Then when the method calls occur, the summaries which we have stored before can be re-used to get the alias information without re-analyzing the method. Due to the existence of method calls, here we introduced a new set to gather the alias information caused by method calls.

`CALL_Alias`(B): the set of aliases which is caused by method calls in block B.

In this paper we focus on the programs which have condition branches, and we use the flowing equation to compute `OUT_Alias`$[B]$:

$$OUT\_Alias[B] = (IN[B]\_Alias - KILL\_Alias[B]) \bigcup GEN\_Alias[B] \bigcup CALL\_Alias[B]) \quad (3)$$

4. Case Study

A control flow graph (CFG) is a directed graph $CFG = (N,E,entry,exit)$, where N is a set of nodes, each node represents a basic block (i.e., a straight-line piece of code without any jumps or jump targets); E is a set of edges which are used to denote jumps in the control flow, *entry* and *exit* are the entry block and exit block, respectively. The entry block is the block through which control enters into the CFG, while the exist block is one through which all control flow leaves the CFG.

```
class Example{
        int a;
        public int getA() {return a;}
        public void setA(int a) {this.a = a;}
}
public class Program {
        public static void changeValue(Example e){
 C[1]    e.setA(10);
        }
        public static Example returnExample(){
 C[2]           Example ee = new Example();
                ee.setA(2);
                return ee;
        }
        public static void main(String[] args) {
           int x=1,y=2,z=5,m=3,n;
           Example ex1 = new Example();
 B[1]      Example ex2 = new Example();
           ex1.setA(x);ex2.setA(y);
           n = new java.util.Random().nextInt(10)
              if(n<3){
 B[2]            ex1 = ex2;
                 ex1.setA(m);
                 if(ex2.getA() == 2){
 B[3]                    z = m-1;
                 }else{
 B[4]                    z = m+1;
                 }
 B[5]       }else if(n<7){
 B[6]             changeValue(ex1);
                 if(ex1.getA() == 1){
 B[7]                    z = m-2;
                 }else{
 B[8]                    z = m+2;
                 }
            }else{
 B[9]          ex1 = returnExample();
               if(ex1.getA() == 2){
 B[10]             z = m+10;
               }else{
                   z = m-10;
 B[11]         }
            }
 B[12]  System.out.println(z);
        }
}
```

Fig. 2. Motivating Example for The Program P.

We use the Java program shown in Fig. 2 to illustrate our approach. This program is extracted from a large program and it includes alias information and references escape. The alias information is caused by references assign and the two kinds of references

escape which we mentioned above. As control flow graph is a directed graph, we firstly identify all of the basic block, i.e., all straight-line pieces of code without any jumps or jump targets. Then, according to the jump between the basic blocks, we can get the control flow graph of this program as shown in Fig. 3, which we got by means of control flow analysis for the partial Java program as presented in Fig. 2.

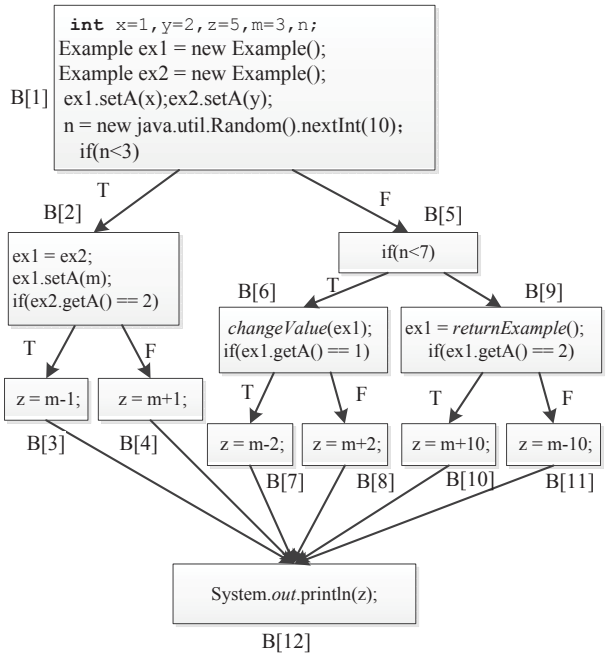

Fig. 3. Control Flow Graph of The Example Program *P*.

In this paper, we use program paths [9] to describe the control flow information. Program paths are an approach to presenting a dynamic control flow of a program that can capture the complete control flow information of a program in a compact and tractable form. So, we can use the program path to describe the entire control flow of the program to capture a complete dynamic behavior of the program. Program paths here present the basic block sequences of the programs. They include executable paths and infeasible paths. Each program path starts from the entry block $B[1]$ and ends at the exit block $B[12]$. So, by traversing the whole control flow graph (see Fig. 3) , we can get the whole program paths of our program *P* as listed in Table 1.

Table 1. All of the program paths.

No.	Program Path(s)
1	B[1] - B[2] - B[3] - B[12] - exit;
2	B[1] - B[2] - B[4] - B[12] - exit;
3	B[1] - B[5] - B[6] - B[7] - B[12] - exit;
4	B[1] - B[5] - B[6] - B[8] - B[12] - exit;
5	B[1] - B[5] - B[9] - B[10] - B[12] - exit;
6	B[1] - B[5] - B[9] - B[11] - B[12] - exit;

4.1. Data Flow Analysis

In this section, we use traditional data flow analysis and ignore the alias information and the influence of method calls. The `def-use` information, including $Out[B]$ and $Use[B]$ of each block, is computed by the methods which we mentioned in section 3. Then, after identifying the basic blocks of the program, we can use the `def-use` information to get the control flow information. This means that we can get all of the program paths and determine the program paths whether are executed or infeasible.

In the control flow graph, the control flow of the program is determined by the value of the predicate expression in the condition branches. In order to get the control flow information, we should use the def-use information to decide the value of predicate expression in each condition branches. First, we analyze the def-use information of variable n in condition branch statement "`if`$(n < 3)$", after querying the set $Out[B]$ and $Use[B]$ of each block, we find that the variable is defined in block $B[1]$ and only used in the same block. The value of variable n is a random number between 1 and 10. Therefore, the value of the condition branch statement cannot be identified. In this situation, both the truth branch and the false branch of the block $B[1]$ can be regarded as feasible ones which can be reached. Similarly, for condition branch statement "`if`$(n < 7)$" in block $B[5]$, since the value of variable n is a random number between 1 and 10, we regard both the `true` and `false` breach as executable ones.

Then we analyze the condition branch statement "if(ex2.getA() == 2)" in block $B[2]$, the type of the variable *ex*2 is reference types. After searching the

sets $Out[B]$ and $Use[B]$ of each block, it is easy to find that the reference $ex2$ is defined in block $B[1]$ and then $ex2$ is used to change the state of the object which it point to in block $B[1]$. The reference $ex2$ is also used in block $B[2]$, but it does not change anything. Therefore, we can conclude that the value of the predicate expression "ex2.getA() == 2" is true, and the truth branch of block $B[2]$ is executable and the false branch can never be reached. For the condition branch statement "if(ex1.getA() == 1)" in block $B[6]$, the variable $ex1$ is also a reference, and it is both defined and used in block $B[1]$. Therefore, the value of the predicate expression "ex1.getA() ==1" is true, the truth branch of the block $B[6]$ is executable and the false branch can never be reached. For the condition branch statement "if(ex1.getA() ==2)" in block $B[9]$, we can use the result of block $B[6]$. After analyzing block $B[6]$, the value of predicate expression "ex1.getA() ==1" is true, so the value of predicate expression "ex1.getA() ==2" is `false`. Therefore, the truth branch of the block $B[9]$ can never be reached and the false branch is executable.

According to the above analysis of the program P, the control flow information of program P can be displayed as Table 2.

Table 2. The feasibility for each program path using traditional program analysis methods.

No.	Program Path(s)	Feasibility
1	B[1]-B[2]-B[3]-B[12]-exit;	executable
2	B[1]-B[2]-B[4]-B[12]-exit;	infeasible
3	B[1]-B[5]-B[6]-B[7]-B[12]-exit;	executable
4	B[1]-B[5]-B[6]-B[8]-B[12]-exit;	infeasible
5	B[1]-B[5]-B[9]-B[10]-B[12]-exit;	infeasible
6	B[1]-B[5]-B[9]-B[11]-B[12]-exit;	executable

4.2. Our Method

Here, we take the alias information and the influence of method calls into account and we regard $C[1]$ and $C[2]$ as two particular basic blocks. The def-use information, including $Out[B]$ and $Use[B]$ of each block, is computed by the method mentioned in subsection 3.1 of section 3, and the alias information, including $CALL\_alias[B]$, $GEN\_alias[B]$, $OUT\_alias[B]$, is computed using the method mentioned in subsection 3.2 of section 3. Then, we can use the def-use information and the alias information to get the control flow information. That means we can get all program paths and determine the program paths which are executable or infeasible.

For condition branch statement "`if`$(n < 3)$" in block $B[1]$ and "`if`$(n < 7)$" in block $B[5]$, since there is no alias information in $OUT\_alias[1]$ and $OUT\_alias[5]$, the analysis result of the two blocks is similarly to the result in previous part. As for the value of predicate expression "$n < 3$" and "$n < 7$" cannot be determined. Therefore, both branches of block $B[1]$ and $B[5]$ are executable.

For condition branch statement "if(ex2.getA() == 2)" in block $B[2]$, after analyzing the alias information and def-use information of the block, we find that it generates an alias pair $\langle ex1, ex2\rangle$ in this block, the alias pair is in set $OUT\_alias[2]$. Then the reference $ex1$ and the reference $ex2$ are aliases and they point to the same object. Since the reference $ex1$ is in the set $Use[2]$, the statement "ex1.set(m)" changes the state of the object which $ex1$ points to, the reference $ex2$ is influenced by this change. Therefore, the value of predicate expression "ex2.getA() == 2" is `false`. The false branch of block $B[2]$ is executable and the truth branch will never be reached. For the condition branch statement "if(ex1.getA() == 1)" in block $B[6]$, it is easy to find that there is an alias pair $\langle ex1, e\rangle$ in $CALL\_alias[6]$ and also in $OUT\_alias[6]$. This alias pair is caused by calling the method $chageValue()$ and passing the reference $ex1$ to it. Then references $ex1$ and e point to the same object. Since the reference e is in the set $Use[6]$ and it is used in the statement "e.set(10)" in method body $C[1]$, the state of the object which e points to has changed, the variable $ex1$ is influenced by this change. Therefore, the value of predicate expression "ex1.getA() == 1" is `false`, the `false` branch of block $B[6]$ is executable and the truth branch could never be reached. For the condition branch statement "if(ex1.getA() == 2)" in block $B[9]$, it is easy to find there is an alias pair $< ex1, ee >$ in $CALL\_alias[9]$ and $OUT_alias[9]$. This alias pair is caused by calling the method $returnExample()$, af-

ter querying the def-use information of the method body $C[1]$, we find that reference *ee* is defined and used in $C[1]$, the method returns reference *ee* to its caller *ex*1 at the end of the method. Therefore, it is obvious that the reference *ex*1 does not point to the object which is defined in block $B[1]$. In block $B[9]$, the variable *ex*1 and *ee* are aliases and they point to the same object. So we can conclude that the value of the predicate expression "ex1.getA() == 2" is `true`, and the truth branch of block $B[9]$ is executable and the false branch will never be reached.

According to the above analysis of the program *P*, the control flow information of program *P* can be achieved as shown in Table 3.

Table 3. Feasibility for each program path using our method.

No.	Program Path(s)	Feasibility
1	B[1]-B[2]-B[3]-B[12]-exit;	infeasible
2	B[1]-B[2]-B[4]-B[12]-exit;	executable
3	B[1]-B[5]-B[6]-B[7]-B[12]-exit;	infeasible
4	B[1]-B[5]-B[6]-B[8]-B[12]-exit;	executable
5	B[1]-B[5]-B[9]-B[10]-B[12]-exit;	executable
6	B[1]-B[5]-B[9]-B[11]-B[12]-exit;	infeasible

4.3. Analysis

For Table 2, since we analyze the program using the traditional methods without considering alias information and the influence of method calls, we get the results that program paths, like No.1, 3, 6, are executable, and other program paths are infeasible. For Table 3, we take alias information and the influence of method calls into account, we get the results that program paths, like No.2, 4, 5, are executable, and other program paths are infeasible. After comparing the result of Table 2 and Table 3, we can conclude that, since the traditional data flow analysis does not consider alias information and the influence of method calls, it may miss some important data information. As we can see in the Table 3, since we consider alias information and the influence of method calls, the program paths which are executable in traditional data flow analysis may become infeasible, and the program paths which are infeasible in traditional data flow analysis may become executable. Consequently, we can find that the alias and reference escape can influent the control flow of the program. The the program is analyzed, the alias and reference escape must be considered.

5. Related Work

As an important technology of program analysis, data flow analysis is widely applied into kinds of programs. As for some programming languages have their own features, such as variable scope, exception-handling constructs, pointer alias and references escape. These features give a facility and convenient way for the developers to implement the functions of the program. However, if they are not considered during data flow analysis, the analysis results may be influenced.

The variables may be hidden and covered for their scope in a program, if this situation is not taken into account in traditional data flow analysis, we will get inaccurate data flow information. An improved data flow analysis method based on variables scope and traditional dataflow analysis method was proposed to solve the problem of variables being hidden and covered in C program language [6]. For data flow analysis of C++ and Java programs to be correct and precise, the flows induced by exception propagation must be properly analyzed.

Zhang and Jiang et al. [10] adopt a static analysis approach to detecting infeasible paths of programs with exception-handling constructs. In their case study, they only considered the different conditional statements of C++ program, and the pointer alias and reference escape are not taken into account.

As for Java programming language, it also integrates many useful features of modern languages, such as alias, reference escape, exceptions and so on. furthermore, exceptions are also widely used in Java programs which pose new challenge to developers to have data flow analysis. Shelekhov and Kuksenko [11] proposed a method to analyze the exception handling in java programs. The control flow structures for analysis of exception handling are constructed using the information of data flow analysis. However, the alias and reference escape are not considered in their work.

In some new approaches, the implicit control

flow for a raised exception is represented explicitly. Exception branches, exception plateaus, and exception exits for methods and method calls are introduced as additional control flow structures for analysis of exception handling [7,11,12]. Data flow analysis can be classified into two categories: flow-sensitive and flow-insensitive. A flow-sensitive interprocedural pointer alias analysis algorithm and a flow-insensitive interprocedural pointer alias analysis algorithm were presented to improve the efficiency of alias analysis [5,12]. References escape could only occur in Java programs, the data flow information may be hard to obtain when analyzing a program with references escape. Some new program analysis methods which considering escapes analysis are proposed to get more accurate data flow information [3,13,14]. The paper [3] combined pointer and escape analysis for Java programs, and corresponding algorithm was given out. The complete or incomplete Java programs can be analyzed using this method. While in the paper [13], they gave a algorithm for escape analysis of Java objects, and a program abstraction for escape analysis was introduced, using connection graph to build reachability relationships between these objects and references, which can be used to identify the non-escaping objects. And the paper [14] presents a escape analysis for specific runs of Java programs. All possible escape objects at runtime are tracked and which object will escape is determined.

Blanchet [15] used two interdependent analysis, one forward and one backward, for the design and correctness proof of escape analysis for Java. And a method was introduced to prove the correctness of escape analysis.

In this paper, we give out a static approach to detecting control flow information and data flow information of Java programs, and the alias and references escape are considered.

6. Conclusions and Future Work

How the alias and references escape influence on Java program analysis is a challenging work. In this paper, we have proposed an extension approach to the traditional data flow analysis to analyze Java programs. In our method, the alias information and references escape are taken into account. We show how implicit alias information occurs when calling a method and the way the alias and references escape influence the control flow information of the program. The analysis of our method is performed in two steps: 1) compute the def-use information and alias information of each basic block in control flow graph. 2) Use the information to analyze the program and get control flow information. The analysis results of the example we give out show that our method can detect the control flow information and demonstrate the influence of alias and references escape on Java program analysis. From the analysis of the alias and reference escape in our program, we can achieve practically all of the feasible and infeasible program path. Consequently, using our proposed method, according to the analysis results of the specific programs, based on the feasible program paths, we can give out the test data. By this means, the test cases can be obtained. So, this method can reduce the quantity of test cases.

However, our approach could only be used in the programs with branches. And we only consider the part of the features of Java programs, such as alias and reference escape. Our future work is to verify the effectiveness of our approach and extend our work to analyze different kinds of Java programs with more features.

Acknowledgements

This work was supported by the National Natural Science Foundation of China (NSFC) (61073050, 61170044), Shanghai Leading Academic Discipline Project (J50103). Key Laboratory of Science and Technology Commission of Shanghai Municipality under Grant No. 09DZ2272600. The authors are grateful to the anonymous referee for a careful checking of the details and for helpful comments that improved this paper.

References

1. F. E. Allen, "Control flow analysis," *SIGPLAN Not.*, **5**(7), 1–19 (1970).

2. N. D. Jones, "Flow analysis of lambda expressions (preliminary version)," *Proc. of the 8th Colloquium on Automata, Languages and Programming*. London, UK, UK: Springer-Verlag, 114–128 (1981).
3. J. Whaley and M. Rinard, "Compositional pointer and escape analysis for java programs," *Proc. of the 14th ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA'99, New York, NY, USA: ACM, 187–206 (1999).
4. R. Chen, *Infeasible path identification and its application in structural test (In Chinese)*. Doctoral Dissertation, Institute of Computing Technology of Chinese Academy of Sciences, China (2006).
5. M. Hind, M. Burke, P. Carini, and J.-D. Choi, "Interprocedural pointer alias analysis," *ACM Trans. Program. Lang. Syst.*, **21**(4), 848–894 (1999).
6. S. Jiang and X. Zhao, "Data flow analysis based on the variable scope (In Chinese)," *Computer Science*, **39** (3), 131–134 (2012).
7. A. Stone, M. Strout, and S. Behere, "May/Must analysis and the DFAGen data-flow analysis generator," *Information and Softw. Technol.*, **51** (10), 1440–1453 (2009).
8. Y. Zhang, S. Jiang, Q. Wang, and X. Zhao, "Static approach to detecting infeasible basis paths (In Chinese)," *Journal of Frontiers of Computer Science and Technology*, **6** (2), 144–155 (2012).
9. J. R. Larus, "Whole program paths," *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, ser. PLDI'99. New York, NY, USA: ACM, 259–269 (1999).
10. Y. Zhang, S. Jiang, Q. Wang, and X. Zhao, "Infeasible basis paths detection of program with exception-handling constructs," *IJACT: Int. J. of Advancements in Comput. Technol.*, **4** (1), 492–503 (2012).
11. V. I. Shelekhov and S. V. Kuksenko, "Data flow analysis of Java programs in the presence of exceptions," *Proc. of the 3rd Int. Andrei Ershov Memorial Conf. on Perspectives of System Informatics*, ser. PSI'99. London, UK, UK: Springer-Verlag, 389–395 (2000).
12. S. Jiang, B. Xu, and L. Shi, "An approach of data-flow analysis based on exception propagation analysis (In Chinese)," *Journal of Software*, **18** (4), 832–841 (2007).
13. J. D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff, "Escape analysis for Java," *Proc. of the 14th ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA'99. New York, NY, USA: ACM, 1–19 (1999).
14. K. Lee, X. Fang, and S. P. Midkiff, "Practical escape analyses: How good are they?" *Proc. of the 3rd Int. Conf. on Virtual Execution Environments*, ser. VEE'07. New York, NY, USA: ACM, 180–190 (2007).
15. B. Blanchet, "Escape analysis for Java^{TM}: Theory and practice," *ACM Trans. Program. Lang. Syst.*, **25** (6), 713–775 (2003).