Dead Code Elimination

Removal of unreachable methods and dead if conditions in Android applications

Tadasa Dayanidhi Mantri
Technische Universität Darmstadt
Darmstadt,Germany
tadasa_dayanidhi@stud.tu-darmstadt.de

ABSTRACT

Code that is unreachable or that does not affect the program can be eliminated. Many applications might have dead code inserted by application developer ignorantly or while testing.The approached used by us is inter-procedural. It implemented by removing the unused methods and the dead if conditions using soot framework.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Theory

Keywords

soot, inter-procedural,constant propagation

1. INTRODUCTION

Dead code elimination is an optimization for removing code which does not affect the program results. It helps in shrinking the program size and allows a running program to avoid executing irrelevant operations. Unreachable code is considered as dead code.Thus dead code might lead to unnecessary memory consumption and execution issues.Dead code could also lead to false positives for e.g. taint analysis identifying dead code as a sink.Conditionally unreachable code and unused functions are removed in our approach in Soot framework.
Soot provides unreachable code elimination. However the available implementation is intra-procedural. We have developed inter-procedural analysis for dead code elimination. This approach uses VASCO framework for inter-procedural constant propagation which helps in determining if a used "ifcondition" yields constant value due to the use of constants in condition.

ICA LAB '15 Darmstadt, Germany

The approach is implemented in two steps. The first one being removed the unused methods of the application. A Call graph of an application would provide directed flow representing the relationships between subroutines.Hence we would get a list of all the methods that are being used in the application. We calculate the unused methods by checking all methods which are present in Application and not added in Call Graph. These unused methods are dead code and hence can be removed. The second step involves detecting the "if conditions" which result in constant value. VASCO framework is being used by us for constant propagation.It performs precise inter-procedural data flow analysis using Value Sensitive Contexts. Hence we have a set of constant values being propagated in the program. If these constant values are used in "if conditions" it would yield a constant result. So the un-executed part of the condition can be deleted.

```
a=5;
if(a==5)
        Print("constant");
else
        Print("not constant")
```

For e.g. if for a condition explained above result yields "true" then the false part is never executed and hence we can remove the statements in else part.
The next section explains what implementation is done by me for this project.

2. IMPLEMENTATION PERFORMED:

I handled the dead ifs removal part for the implementation. Here we use Vasco implementation to check if the variables that are being used in the if condition are already propagated and represent constant values and the resultant operation in if condition is constant then this if condition is reported to have a constant value in each execution. Vasco helps in constant propagation of values. So at the execution of every statement we consider the outset containing the constants, of all the predecessors and hence we have a constant propagation of variable in the program till the current statement. Vasco implements inter-procedural constant propagation by mapping the parameters and return values of the called method. However I came across a situation wherein the parameter that is being passed in the called method and the return value of the method are represented by the same variable in jimple.

```
int q=getq(5);
this.p=getp(q)
```

```
if(p is constant)
        print true
else
        print false
```

The line "this.p=getp(q)" is displayed as:

```
$i0=call getp($i0);
```

In this case , $i0 is assigned null before recieving the output of the function since its a function call. However from the function getp() when a value is returned, the merge of in(values propagated from above) and the current Outset do not have the same value and hence $i0 is assigned null. We handled this case by implementing checkInOutSet() to merge the outset of the predecessor and the current inset value. Execution implemented in our analysis.However we need to call this function from Vasco.We would be attaching the Vasco framework code with the project.
If removal is done based on the condition assessed. If the result condition is true then I replace the if statement with goto expression of the if statement i.e. the If statement is removed and and we only have the direct goto statement. In case if the result is false then we remove the whole if statement and hence the Label with Goto for this condition doesn't have any predecessor. So this could easily be removed when we call unreachable code eliminator functionality of Soot.
I have also implemented the method removal part for handling the special case of interfaces and Dynamic invocations of runtime methods. When a class implements an interface all of its functions needs to be implemented. Hence if suppose our class implements that interface and has the definition of all the implemented methods, it might not call all of the implemented methods of interface. Hence this method would not be detected in callgraph and would be marked as unused. Hence we need to specifically handle this case where the parent class of a method implements a interface none of the implemented methods should be deleted.

3. CONCLUSIONS

We have tested the code on some android applications and plotted graph with the results. We could confirm that many of the applications contain dead code. The dead code removal helps in better performance in static code analysis.