Efficient online detection of dynamic control dependence

Debjeet Majumdar
Dept. of Computer Science,IIT Kanpur
M.tech.-15111014
debjeetmaj@cse.iitk.ac.in

Swapnil Mhamane
Dept. of Computer Science,IIT Kanpur
M.tech.-15111044
mswapnil@cse.iitk.ac.in

ABSTRACT
This project introspects the idea for efficient online detection of dynamic control dependence presented in the paper by Bin Xin et al. [8]. The previous techniques were incapable to capture some properties of dynamic control dependence, as they were simply runtime translation of their static counterpart. The paper proposes a new definition of dynamic control dependence and discusses the algorithm for it. In our project we implement this new algorithm for intra-procedural and regular inter-procedural cases.

1. SCOPE
Our implementation analyses java programs , which uses the soot[3] at lower level and aspectj[1] at higher level to implement the algorithm discussed in the paper. Our analysis restricts detection of dynamic control dependence in intra-procedural case and inter-procedural case with regular control flow.The implementation also focuses in using an aspect oriented approach for our analysis.The aspect oriented approach helps us to work at a higher level of abstraction.

2. INTRODUCTION
Control dependence can be informally defined as the dependence of a statement s on a predicate p, which decides that s gets executed.Thus we are capturing program behaviour with respect to a predicate statement. This is called static control dependence.Dynamic control dependence gives the runtime effect of the predicate instance on each statement instance.

Static analysis reveals the dependence of statements on a predicate but is an over-approximation as the actual execution path is unknown. Dynamic analysis in contrast is a precise analysis but can only report dependence actually seen in an execution path, so may not be safe.So the precision aspect is required in many such applications of dynamic control dependence eg.dynamic slicing[5], dynamic information flow[7] ,data lineage[6],compiler optimizations[4].

3. PAPER SUMMARY
The paper formally defines dynamic control dependence and builds an efficient algorithm to find it in all cases.It talks about an online strategy to find the dependence. There are two popular algorithm strategy in finding dynamic control dependence: *online* and *offline*. Offline approach involves collecting execution traces and backward tracing them to reveal the dependence. The execution traces can be of large volumes; also simulating the call stack is tedious. Recursive calls are hard to handle.

The online approach on the contrary couples the analysis alongside the execution of the program. The approach has an advantage over offline counterpart that call stack simulation can be done on the fly. The disadvantage is that coupling causes recompilation and an additional runtime overhead. The authors of the paper[8] claim that the existing algorithms are straightforward translation of the static one. The paper discusses a refined definition of dynamic control dependence which provides a base for efficient algorithm.

3.1 Formal Definition
Dynamic control dependence until now has not been formally defined,lets see its definition

DEFINITION 1. *[**Dynamic Control Dependence**[8]] An execution instance x_i dynamically control depends on another instance y_j if and only if*

1. $y_j < x_i$
2. $x \xrightarrow{scd} y$
3. $\nexists z_k \; s.t. \; y_j < z_k < x_i \wedge x \xrightarrow{scd} z$

Note :

- x_i is the ith instance of statement x
- $y_j < x_i$ means x_i statically executes before y_j

3.2 Intra-procedural Dynamic Control dependence
We first limit our scope to intra-procedural case, and present the new novel definition. The definition is general to both structural and unstructural control flow (*(*break), *(*return)) in this case.

DEFINITION 2. *[**Dynamic post dominance**[8]] A statement instance x_i dynamically post-dominates y_j , denoted by $x_i \xrightarrow{dpd} y_j$ if and only if $y_j < x_i$ and x statically strict post-dominates y, denoted by $x \xrightarrow{spd} y$, and there does not exist x_k such that $y_j < x_k < x_i$*

DEFINITION 3. *[**Dynamic control dependence NEW**[8]] An execution instance x_i dynamically control depends on the largest $y_j < x_i$ if and only if x_i dynamically post-dominates any z_k in between y_j and x_i but not y_j*

It can be shown that the definition 1 and 3 are equivalent. Proof is omitted here.

The below constructs are essential for comprehending the detection algorithm.

1. **Branching Point(BP)**: A statement s which has more than one successor in the CFG. Eg.predicate statements like if conditions.
2. **Immediate Post dominator(IPD(s))** :the immediate post-dominator of a node n is the post-dominator of n that doesn't strictly post-dominate any other strict post-dominators of n.
3. **Control Dependence Stack(CDS)**[8] :a call stack useful to maintain the nested region structure.

DEFINITION 4. *[**Region**[8]] Given an executed instance of a BP s_i, let $IPD(s_i)_m$ be the immediate dynamic post-dominator of s_i, the region directed by s_i, represented by $R(s_i)$, is defined as:*

$$R(s_i) =< (x_j) | x_j < s_i < IPD(s)_m >$$

s_i is called the director of the region.

Properties of Region[8]

1. Given any $x_j \in R(s_i)$ the upper bound of the region, $IPD(s)_m$, dynamically post-dominates x_j
2. Regions in execution are either disjoint or nested.
3. A statement instance x_i is dynamically control dependent on the director of the smallest enclosing region.

Regions provide the base for an efficient detection algorithm

3.2.1 Algorithm

First we statically determine the BPs and the IPDs. The BPs are instrumented with the *Branching()* function call and IPDs are instrumented with the *Merging()* call. The algorithm 1 can be understood with the example in figure 1

In figure 1, for branching points p1 we make a CDS entry. Here the dynamic dependent predicate is the director of the current region, which is given by the topmost entry of the

Algorithm 1 Intra procedural DCD detection algorithm[8]

```
1: procedure BRANCHING(s_i,IPD(s))
2:     if CDS.top().second ≡ IPD(s) then
3:         CDS.top().first = s_i
4:     else
5:         CDS.push(< s_i, IPD(s) >);
6: procedure BRANCHING(t_j)
7:     if CDS.top().second ≡ t_j then
8:         CDS.pop();
```

CDS. In this example we see IPD($p1_1$) =IPD($p2_1$)= 5_1, So we get a nested region, the CDS only keeps track of smaller region amongst the region with same IPD. This optimization is extremely important as, in case of loops the CDS may overflow in absence of this optimization.So there is always just one entry ever on the CDS for n iterations of the loop.

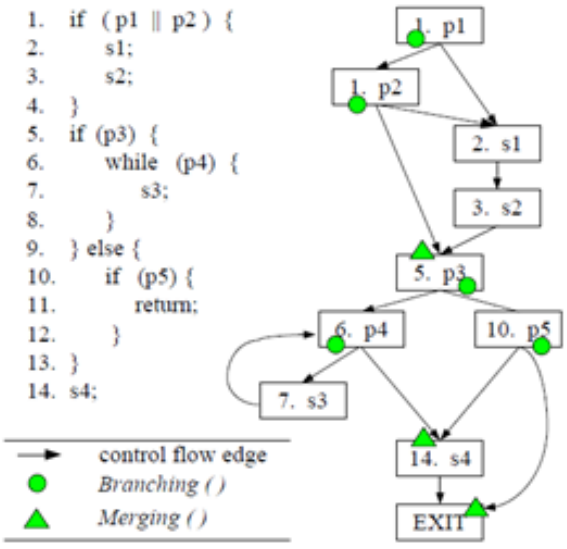

Figure 1: example for intra-procedural dynamic control dependence

3.3 Inter-procedural dynamic control dependence

There are two kinds of inter-procedural control flow regular and irregular flow. The regular flow is normal call and return. The irregular control flow is due to abnormal terminations like exit, exceptions etc. The two cases requires further modification to above algorithm.

Simple approach to inter-procedural control flow is in-lining the called function call. This approach often causes a loss in useful meaning. Analysis in inter-procedural case requires considering the context in which a statement executes. Execution instance of a statement should be accompanied along with different context.

3.3.1 Regular Control flow

A context associated with the method call is called the calling context(CC(s_i)) of the method call. It is an ordered list

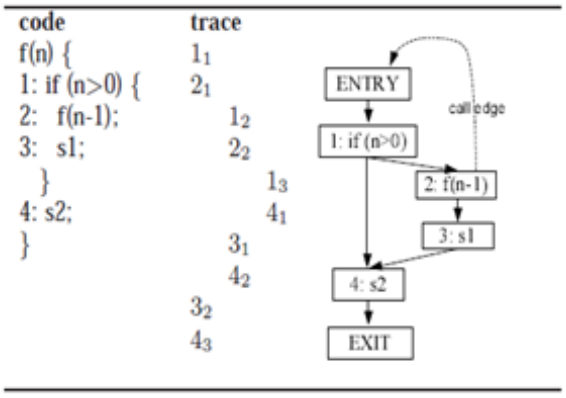

Figure 2: example for regular inter-procedural control flow[8]

of call sites in which s_i is executed. This differentiates the various execution of the same method under different calls. Thus the definition can be refined to check the calling context as well while checking the equivalence of two statement instance. The algorithm is augmented with the additional parameter of calling context. The algorithm handles the recursive programs , which is tedious for its other counterparts previously existent. This can be demonstrated in the following example.

In figure 2, we see a recursive procedure , where without calling context erroneous result will be produced. Here Calling context CC(1_3)=< $2_1, 2_2$ > and CC(3_1)=< 2_1 > In region $R(1_1)$=$(1_1, 4_3)$ we can say $3_2 \xrightarrow{dpd} 1_1$, $1_2 \xrightarrow{dpd} 1_1$ For region R(1_2)=$(1_2, 4_2)$ we can say that,$3_1 \xrightarrow{dpd} 1_2$ If we do not keep track of calling contexts 3_2 and 3_1 will be reported as control dependent on 1_3, which is incorrect.

3.3.2 *Irregular Control flow*

This type of control flow can be caused by exceptions, error handling and exits. The main problem caused to our analysis is the post dominance relationship changes for each statement due to irregular control flow. The following example explains an irregular flow implemented using ***setjmp*** and ***longjmp*** instructions.

Considering an example given in figure 3 we observe that ***longjmp*** at statement 12, alters the control flow directly to statement 2. Thus IPD(p2) changes from s3 to s1 for p2. Hence our previous algorithm fails, which detects the IPD to be within same procedure.

The solution proposed is simple and elegant, we add dummy edge from calling site to ***setjmp***, in the function with ***setjmp***. The calling site here thus becomes a pseudo-predicate.It is although not instrumented with branching call.In procedure with ***longjmp*** instruction or procedure that lead to a call to a procedure with ***longjmp*** instruction, we add a dummy node and add an edge from ***longjmp*** or calling site to dummy node respectively. The dummy nodes are not instrumented, with merging calls. The result of this bypass has been shown in figure 4.

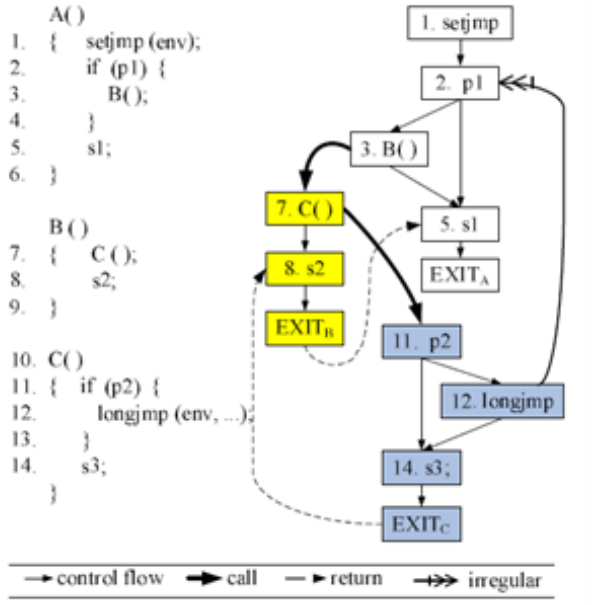

Figure 3: example for irregular inter-procedural control flow[8]

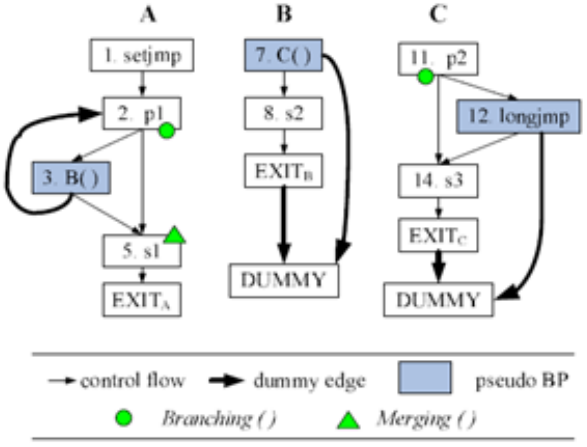

Figure 4: example for handling irregular inter-procedural control flow[8]

4. IMPLEMENTATION SUMMARY

Our project is limited to analysis of JAVA programs. The dynamic detection is implemented using instrumentation of our analysis to the executable JAVA classes. This is done using Soot[3] and AspectJ [1] as our frameworks. The analysis produces recompiled java classes. Then we can execute the instrumented class files to report the dynamic control dependence. The framework can be viewed as follows:

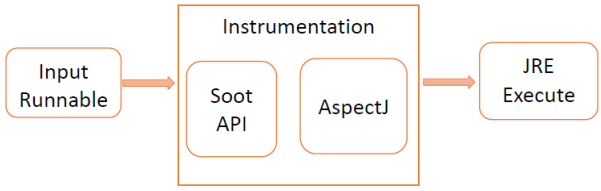

Figure 5: evaluation steps in detection of dynamic control dependence

4.1 Aspect Oriented Programming

Aspect oriented programming is a paradigm that increases the modularity of the program by adding additional behaviour by modifying the current code.Hence we write common code(advice) seperately and add to code. For instance logging calls can be seperately added using aspect oriented software. Thus core functionality can be developed without concernig about maintenance of logs. The aspectJ is aspect oriented software for JAVA. The three pillars in aspect oriented programming :

1. **Pointcuts**: these are interesting locations where we need to modify behaviour, for eg. function calls, object instantiation, field initializations etc.
2. **Joinpoints**: these are set of joinpoints
3. **Advice**: it is action or behaviour that is executed when a specified pointcut occurs.

Although our tool can very well implemented without aspectJ but it enhanced the maintenance of control dependence stack,as well as the evaluation of calling context.

4.2 Passes in DCDAnalyser

The tool developed works in two phases:

1. **Static pre-processing** : This phase uses soot api to detect the branching points and IPDs. Then we plug dummy method calls using soot.These are identified as pointcuts and later weaved by aspectJ compiler with an advice which contains actual *branching()* and *merging()* calls. Also to declare the dynamic control dependence for each statement we instrument log calls to each statement.
2. **Runtime analysis**: The execution of the aspect weaved user program is then done and log calls using log4j[2] simultaneously logs the dynamic control dependence of each statement. The *dcd* log file contains the residue of our analysis, which can be used as required.

4.3 Limitations

- We were unable to implement the irregular inter-procedural control flow, due to shortage of time.Therefore classes with exception handling may not work correctly.
- Also selective analysis of control dependence cannot be done.

5. CONCLUSION

Dynamic control dependence plays a crucial role in several applications where static dependence is not enough and a precise analysis is required. The paper brings up a new novel definition for dynamic control dependence, which provides a more efficient detection algorithm using the concept of region. Our project is a java tool which outlines the detection algorithm as a crosscutting application using aspectj framework.

6. REFERENCES

[1] Aspectj:a aspect oriented software for java analysis.
[2] Log4j, a logging library for java.
[3] Soot: A java optimization framework.
[4] K. J. O. Jeanne Ferrante and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 1987.
[5] B. Korel and J. Laski. Dynamic program slicing.information processing letters. 1988.
[6] X. Z. Mingwu Zhang, Xiangyu Zhang and S. Prabhakar. Tracing lineage beyond relational operators. 2007.
[7] A. C. Myers. Practical mostly-static information flow control. *Proceedings of the 26th ACM Symposium on Principles of Programming Languages (POPL âĂŹ99), San Antonio, Texas, USA, January 1999*, 1999.
[8] B. Xin and X. Zhang. Efficient online detection of dynamic control dependence. *ISSTA*, 2007.