Efficient Online Detection of Dynamic Control Dependence

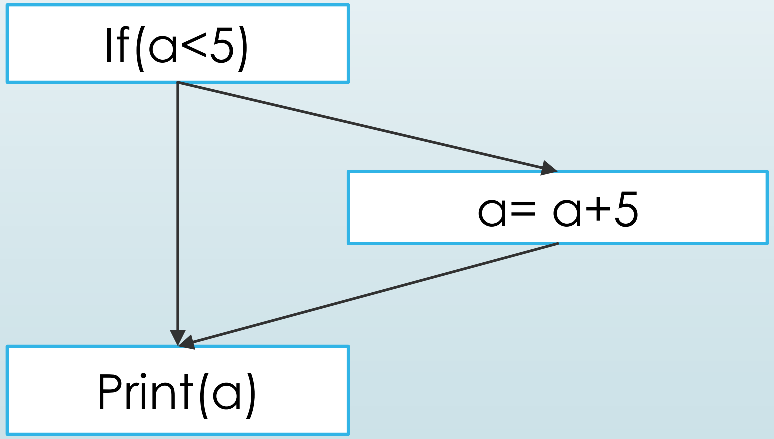
Paper by :-
Bin Xin
Xiangyu Zhang

Presentation By :-
Debjeet Majumdar(15111014)
Swapnil Mhamane(15111044)

Introduction

Control Dependence

- Predicate P : Decision making statement
- Statement s control depends on a predicate statement p



Static and Dynamic Control Dependence

- Statement and Statement instance
- Static Control Dependence (SCD)
  - Control dependence relation between program statement.
- Dynamic Control Dependence (DCD)
  - Control Dependence relation between executing instance of program statement
  - May include statically self dependent statements but different instance dynamically

Applications

- Program slicing
- Information flow analysis
- Data lineage
- Fault Localization

Strategies for DCD

- Offline approach
- Online approach

Offline Algorithm

- Backward traverse the execution trace
- Look for latest predicate statements instance such that $S \xrightarrow{scd} P$
- Drawbacks :
  - Infeasible beyond certain length of trace
  - Difficulty in recursive function handling

Earlier Online Algorithm

- Couples control dependence with call stack maintenance
- Solves problem with recursive function
- Drawbacks :
  - Problems with library functions compiled on different source.
  - Inefficient in handling interprocedural DCD

Defining DCD

- ***Existing definition*** :
  "An execution instance x_i dynamically control depends on another instance y_j if and only if

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

- ***Modified definition*** :

  "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 ."

Important constructs

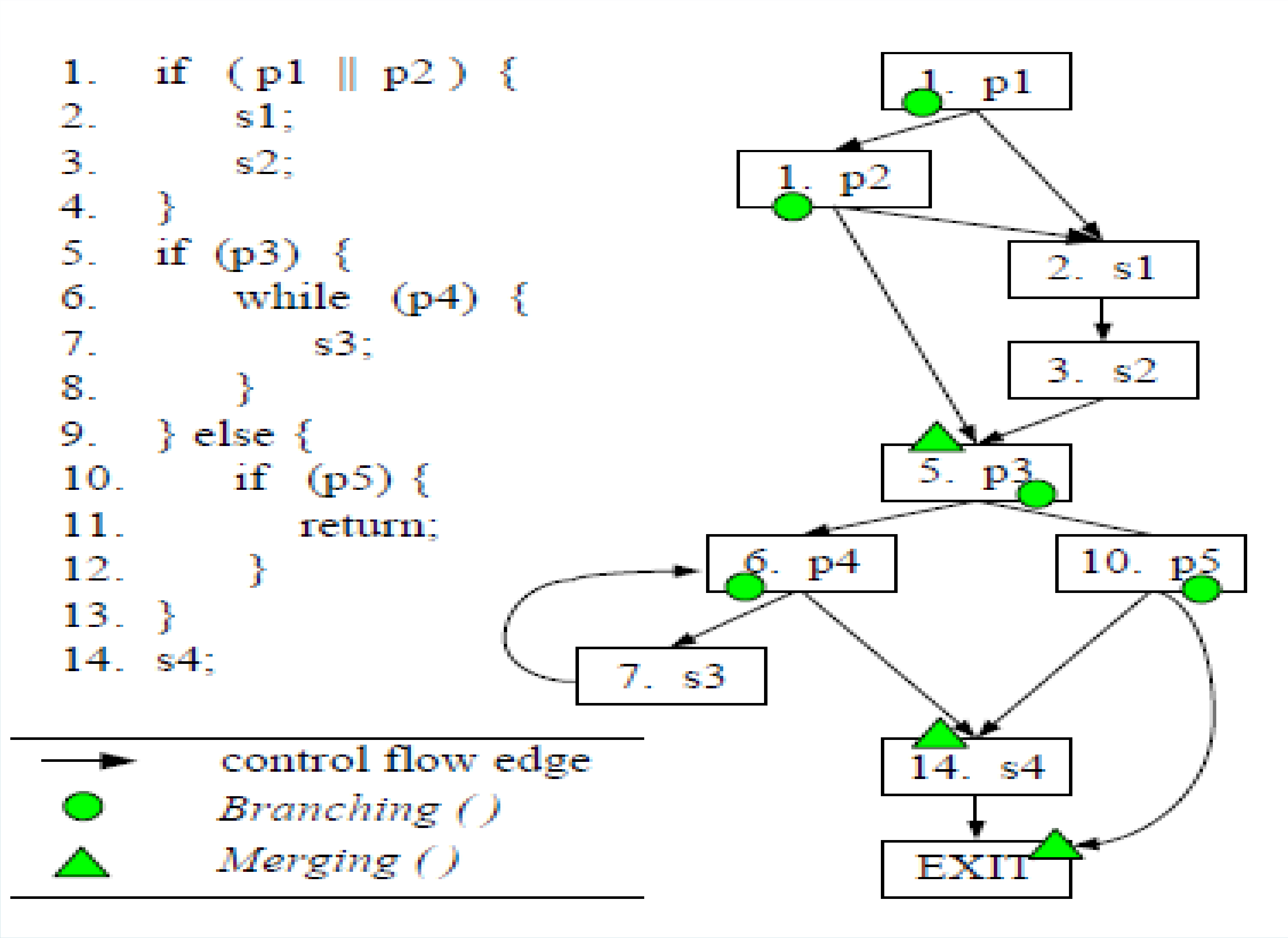
- Branching Point(BP $\hat{s}_i$)
  - More than one successors
- Immediate Post Dominance: IPD$(\hat{s}_i)_m$
- Region
  - $R(\hat{s}_i) = \{X_j \mid \hat{s}_i < Xj < IPD(\hat{s}_i)_m\}$

Intraprocedural algorithm

- **Branching**($\hat{s}_i$, $IPD(\hat{s}_i)$)
  {
  if (CDS.top().second ≡ $IPD(\hat{s}_i)$ { //optimized top entry
  CDS.top().first= $\hat{s}_i$;
  } else {
  CDS.push(< $\hat{s}_i$, $IPD(\hat{s}_i)$ >);
  }
  }

- **Merging** (t_j)
  {
  if (CDS.top().second ≡ t)
  CDS.pop();
  }

Example



1. if (p1 || p2) {
2. s1;
3. s2;
4. }
5. if (p3) {
6. while (p4) {
7. s3;
8. }
9. } else {
10. if (p5) {
11. return;
12. }
13. }
14. s4;
1. p1
1. p2
2. s1
3. s2
5. p3
6. p4
10. p5
7. s3
14. s4
EXIT
control flow edge
Branching ( )
Merging ( )

Cases in Interprocedural dependence

- Function in-lining
  - How would you go about recursive call ?
  - Loss of semantics
  - Solution : Use context sensitive analysis
- Calling context
  - An ordered set of call sites
- Types of interprocedural control flow
  - Regular
  - Irregular

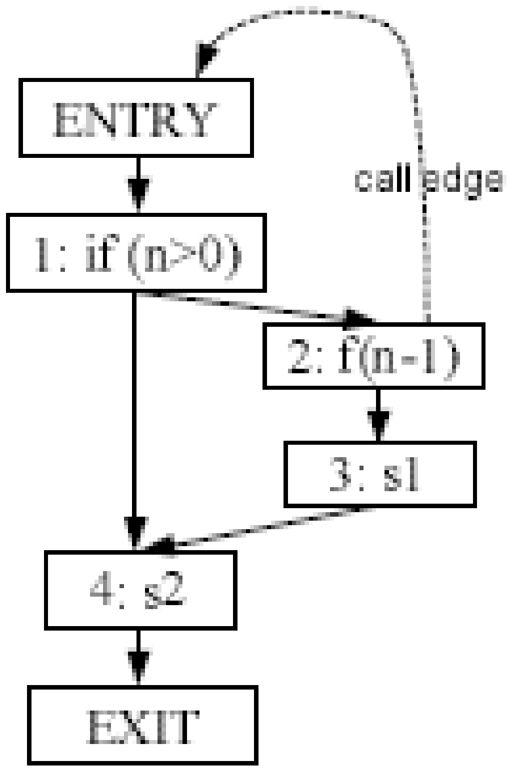
Example

code

```
f(n) {
1: if (n>0) {
2:    f(n-1);
3:    s1;
   }
4: s2;
}
```

trace

1_1
2_1
  1_2
  2_2
    1_3
    4_1
  3_1
  4_2
3_2
4_3



Inter-procedural algorithm

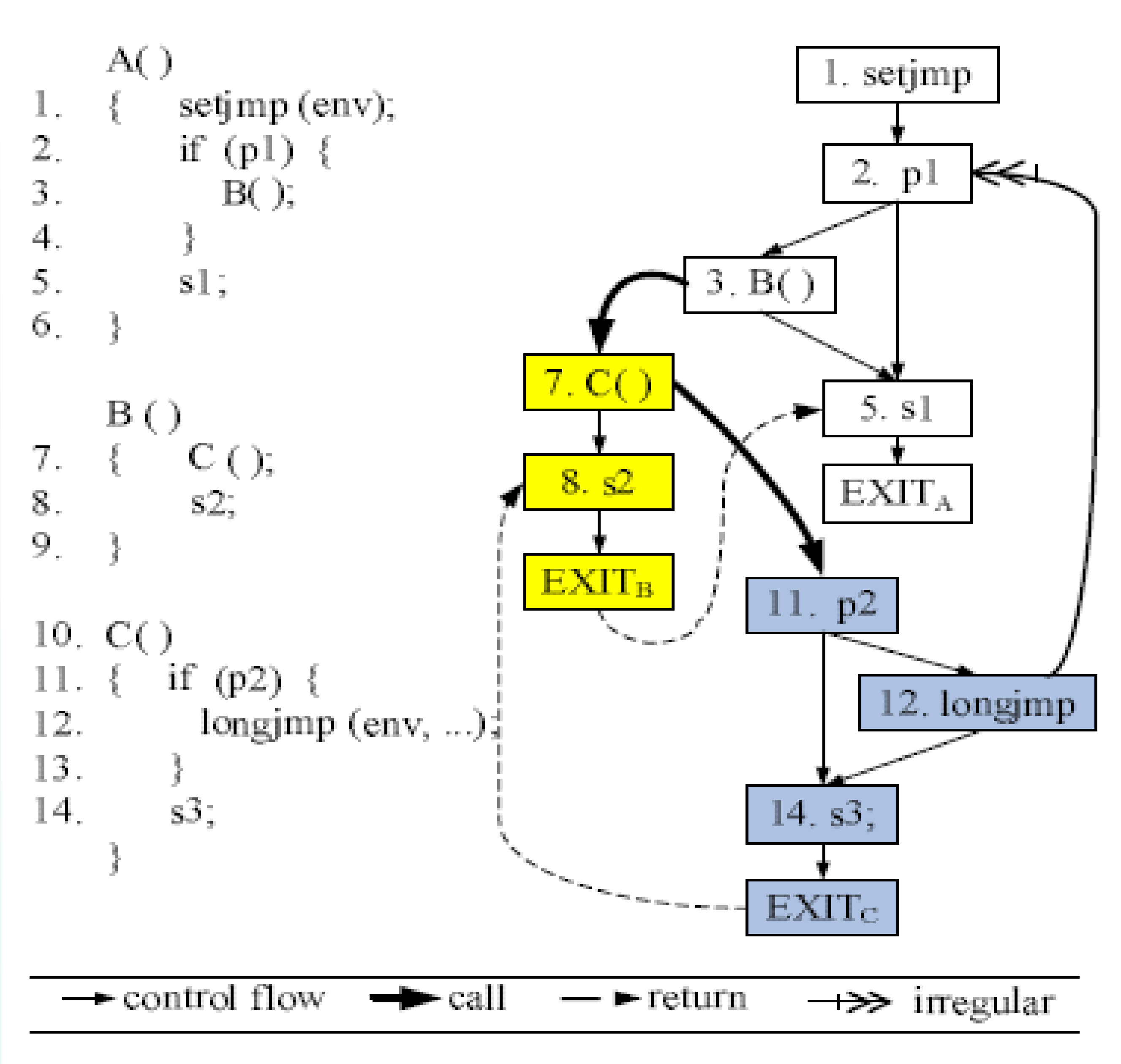
- **Branching**($\hat{s}_i$, $IPD(\hat{s}_i)$, bp)
  {
      if (CDS.top().second ≡ $IPD(\hat{s}_i)^{bp}$ {
          CDS.top().first= $\hat{s}_i$;
      } else {
          CDS.push($< \hat{s}_i , IPD(\hat{s}_i)^{bp} >$);
      }
  }

- **Merging** (t_j , bp)
  {
      if (CDS.top().second ≡ t^{bp})
          CDS.pop();
  }

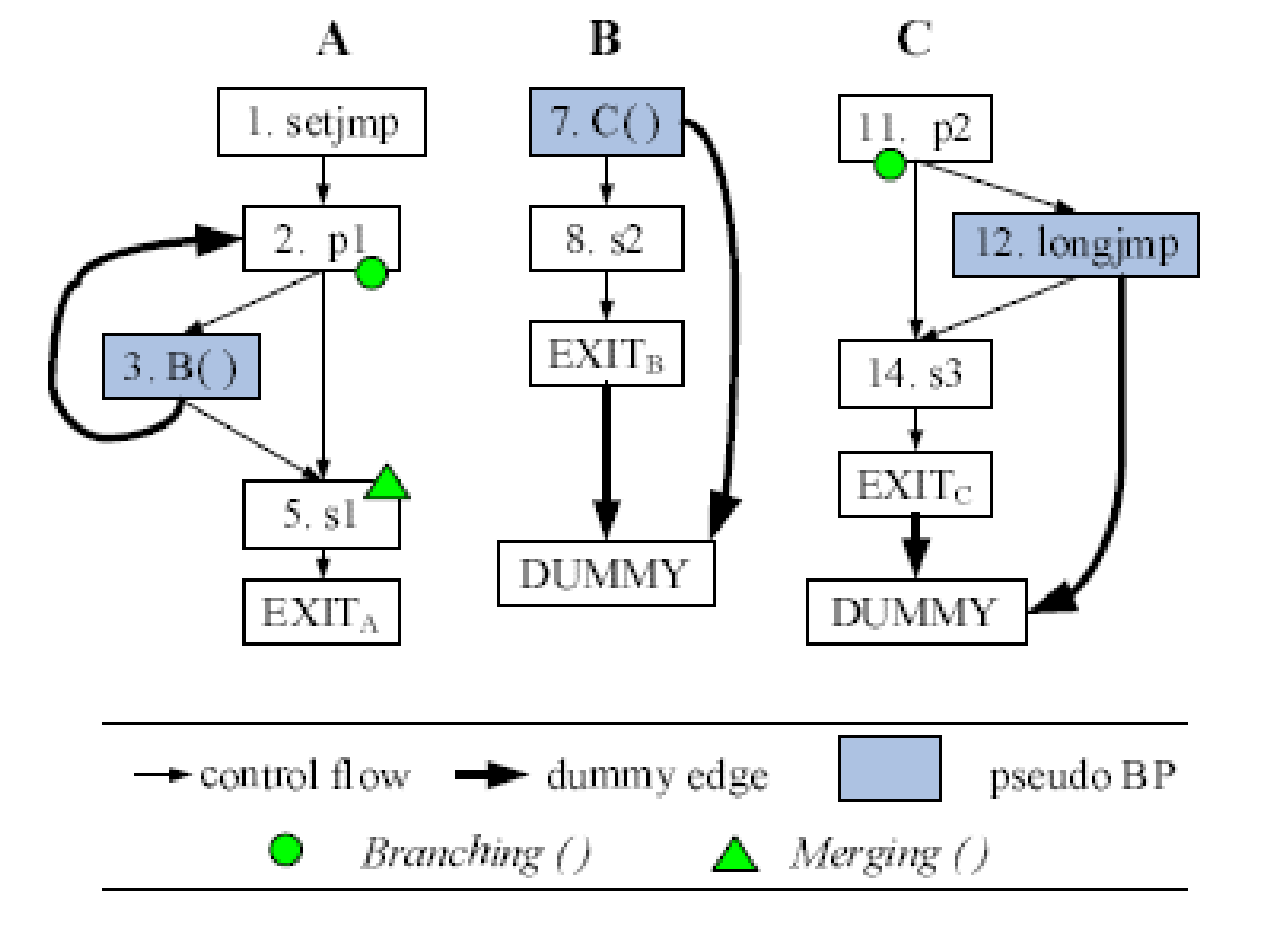
Irregular interprocedural DCD

- Caused by long Jumps , exception handlings etc.
- Solution :
  - Use of dummy nodes and edges
  - Pseudo predicates.

Example

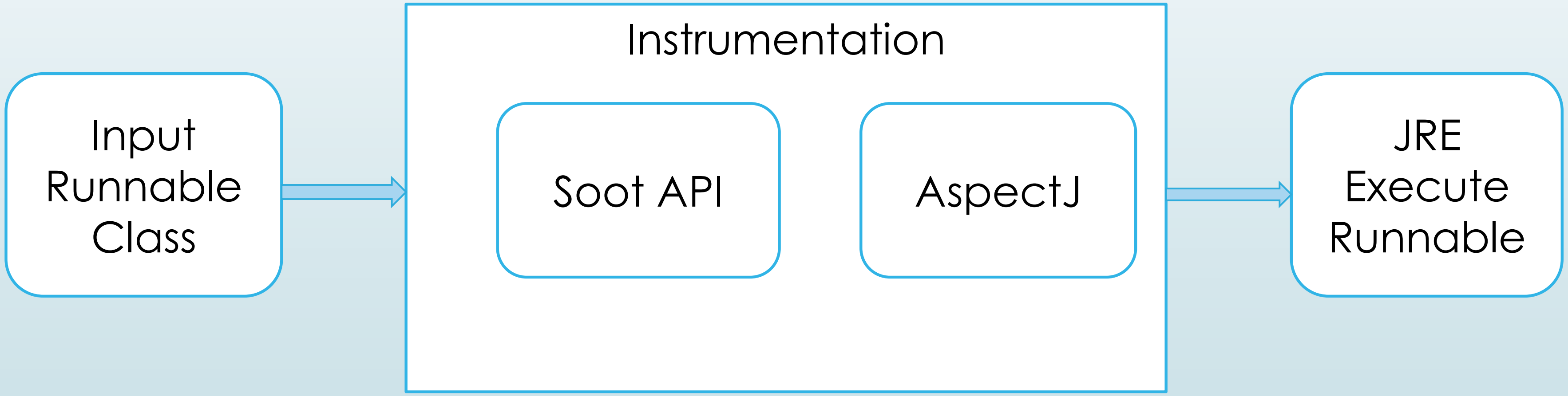

A( )
1. { setjmp (env);
2. if (p1) {
3. B( );
4. }
5. s1;
6. }
B ( )
7. { C ( );
8. s2;
9. }
10. C( )
11. { if (p2) {
12. longjmp (env, ...);
13. }
14. s3;
}
1. setjmp
2. p1
3. B( )
5. s1
7. C( )
8. s2
EXITA
EXITB
11. p2
12. longjmp
14. s3;
EXITC
control flow
call
return
irregular

Modified CFG



Implementation

DCD Evaluation Setup



Implementation

- Dynamic analysis
- Instrumentation
- Soot
  - Find post dominator
  - Statement based instrumentation of function call
- AspectJ
  - Implement algorithm on function call

AspectJ

- Aspect Oriented Programming
  - Separation of cross-cutting concerns
  - E.g. security, logging
- Joinpoints
- PointCuts
  - Joinpoint collection
- Advice
  - Instrument behaviour on collected joinpoints

Conclusion

- Dynamic control dependence plays a crucial role in several applications
- Detection of Dynamic control dependence efficiently is essential
- The paper brings up a new novel definition for dynamic control dependence, which provides a more efficient detection algorithm
- The project outlines the detection algorithm as a crosscutting application using aspectJ

References

- “Efficient online detection of dynamic control dependence” by Bin Xin, Xiangyu Zhang published in ISSTA’07
- X. Zhang, H. He, N. Gupta, and R. Gupta. Experimental evaluation of using dynamic slices for fault location. In *AADEBUG*, 2005.
- “Dynamic Program Slicing” by Hiralal Agrawal, Joseph R.horgan published in conference Programming language design and implementation ’90
- https://eclipse.org/aspectj/
- The Compiler Desing Handbook : Optimization and Machine Code Generation by Y.N. Shrikant, Priti Shankar

THANK YOU …!