Dynamic Program Slicing

Authors: Hiralal Agrawal, Joseph R. Horgan

Presented by:
Casey Casalnuovo

Program Slicing

- Program Slice: Given a point in the program *p* and some variable *x,* what set of statements affect the value of x at p?

- Applications:
  - Verification
  - Maintenance
  - Parallelization
  - **Debugging**

Dynamic Program Slicing

- Static slicing looks at statements that *could* affect *x* at *p* for all possible program inputs.

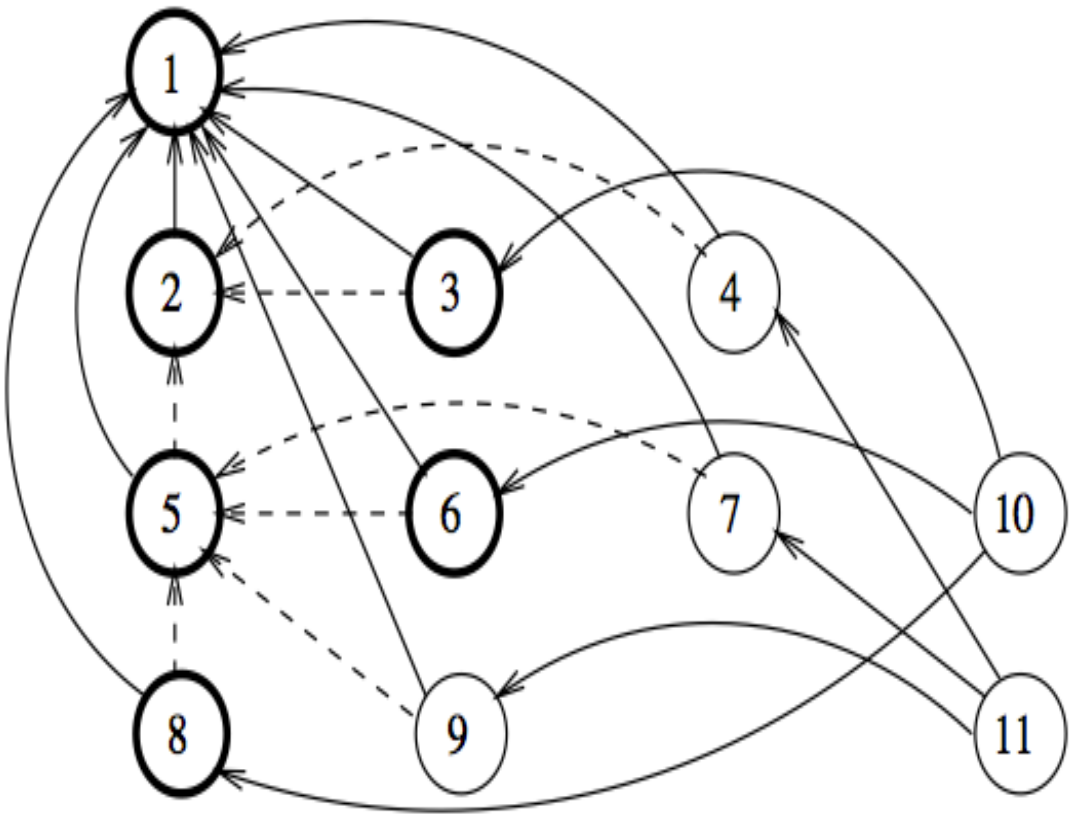
- Dynamic slicing focus on statements that *did* affect the value *for a specific set of inputs.*
  - More useful when trying to determine why a specific test case failed.
  - Paper looks at 4 ways to dynamically slice.

Program Depedence Graph

- Nodes in graph correspond to statements in the program.
- Data-dependence edges
  - Directed edge from one node V1 to another node V2 if V1 depends on a value computed at V2.
- Control-dependence edges
  - Directed edge from V1 to V2 if V1 can immediately follow V2 in the execution path.

```
        begin
S1:         read(X);
S2:         if (X < 0)
            then
S3:             Y := f1(X);
S4:             Z := g1(X);
            else
S5:             if (X = 0)
                then
S6:                 Y := f2(X);
S7:                 Z := g2(X);
                else
S8:                 Y := f3(X);
S9:                 Z := g3(X);
                end_if;
            end_if;
S10:        write(Y);
S11:        write(Z);
        end.
```

Figure 1: Example Program 1




```
        begin
S1:         read(X);
S2:         if (X < 0)
            then
S3:             Y := f1(X);
S4:             Z := g1(X);
            else
S5:             if (X = 0)
                then
S6:                 Y := f2(X);
S7:                 Z := g2(X);
                else
S8:                 Y := f3(X);
S9:                 Z := g3(X);
                end_if;
            end_if;
S10:        write(Y);
S11:        write(Z);
        end.
```

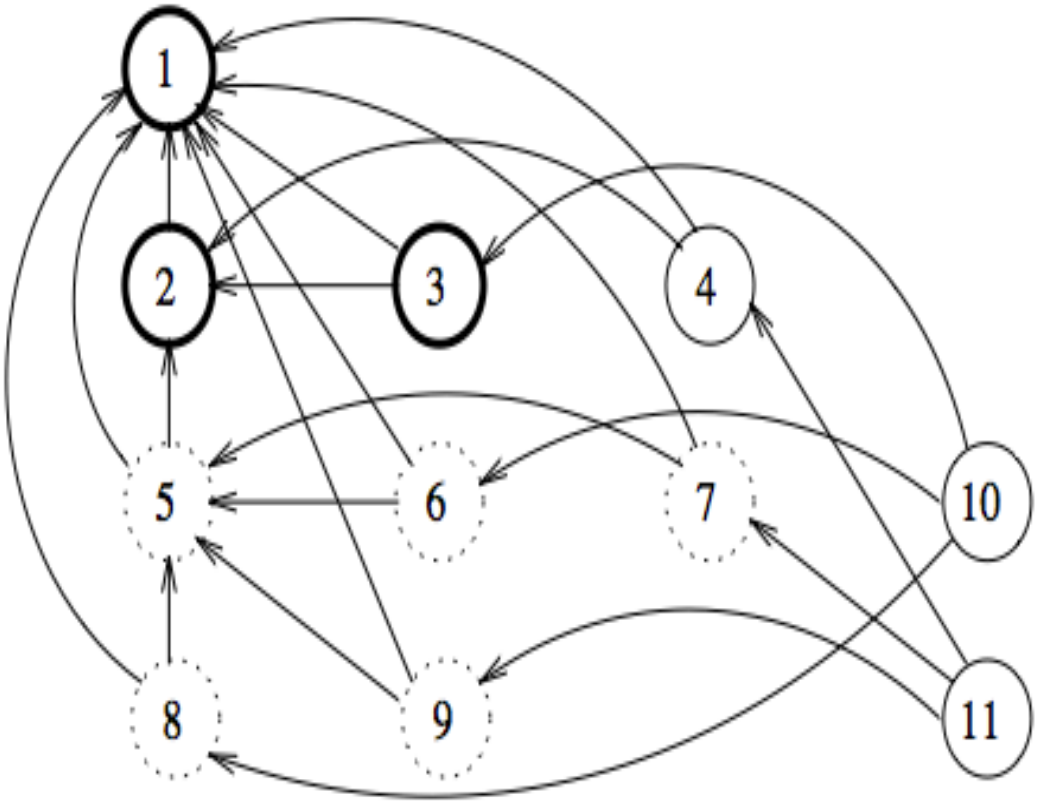
Figure 1: Example Program 1

Figure 2: Program Dependence Graph of the Program in Figure 1. The solid edges denote data dependencies and the dashed edges denote control dependencies. Nodes in bold denote the Static Slice with respect to variable Y at statement 10 in the program.

Slice: What nodes are reachable starting at Node 10?

Dynamic Slicing Approach 1

- Very Simple
- Mark nodes in the Program Dependence Graph that are executed in a given run.
- Run a static slicing algorithm on this subgraph.




```
        begin
S1:         read(X);
S2:         if (X < 0)
            then
S3:             Y := f1(X);
S4:             Z := g1(X);
            else
S5:             if (X = 0)
                then
S6:                 Y := f2(X);
S7:                 Z := g2(X);
                else
S8:                 Y := f3(X);
S9:                 Z := g3(X);
                end_if;
            end_if;
S10:        write(Y);
S11:        write(Z);
        end.
```

Figure 1: Example Program 1

Figure 4: Dynamic Slice using Approach 1 for the program in Figure 1, test-case X = −1, with respect to variable Y at the end of the execution. All nodes are drawn as dotted in the beginning. A node is made solid if it is ever executed; and is made bold if it gets traversed while determining the slice.

Trace Order: <1,2,3,4,10,11>

X= -1

Dynamic Slicing Approach 1

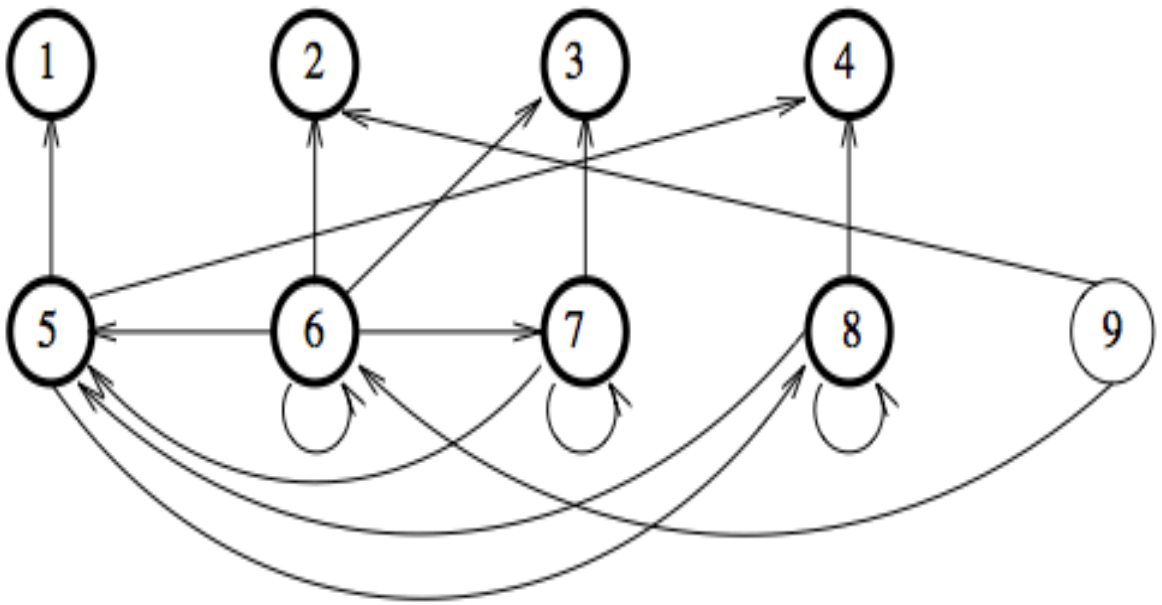
- Very Simple
- Mark nodes in the Program Dependence Graph that are executed in a given run.
- Run a static slicing algorithm on this subgraph.

- Problem:
  - These slices are not always precise.
  - A variable may statically have dependencies on multiple prior statements (Multiple outgoing data dependency edges).

```
        begin
S1:         read(N);
S2:         Z := 0;
S3:         Y := 0;
S4:         I := 1;
S5:         while (I <= N)
            do
S6:             Z := f1(Z, Y);
S7:             Y := f2(Y);
S8:             I := I + 1;
            end_while;
S9:         write(Z);
        end.
```

Figure 3: Example Program 2

Trace order: <1,2,3,4,5^1,6,7,8,5^2,9>
N=1




```
        begin
S1:         read(N);
S2:         Z := 0;
S3:         Y := 0;
S4:         I := 1;
S5:         while (I <= N)
            do
S6:            Z := f1(Z, Y);
S7:            Y := f2(Y);
S8:            I := I + 1;
            end_while;
S9:         write(Z);
        end.
```

Figure 3: Example Program 2

Figure 5: Dynamic slice using Approach 1 for the program in Figure 3, test-case N = 1, for variable Z, at the end of execution. Node 7 should not belong to the slice!

Trace order: <1,2,3,4,5^1,6,7,8,5^2,9>
N=1

Dynamic Slicing 2

- In addition to marking the nodes, mark edges (dependencies) of the graph only as they come up during program execution.
- Find the dynamic slice on this subgraph.

```
        begin
S1:         read(N);
S2:         Z := 0;
S3:         Y := 0;
S4:         I := 1;
S5:         while (I <= N)
            do
S6:             Z := f1(Z, Y);
S7:             Y := f2(Y);
S8:             I := I + 1;
            end_while;
S9:         write(Z);
        end.
```

Figure 3: Example Program 2

Trace order: <1,2,3,4,5^1,6,7,8,5^2,9>
N=1

```
        begin
S1:         read(N);
S2:         Z := 0;
S3:         Y := 0;
S4:         I := 1;
S5:         while (I <= N)
            do
S6:             Z := f1(Z, Y);
S7:             Y := f2(Y);
S8:             I := I + 1;
            end_while;
S9:         write(Z);
        end.
```

Figure 3: Example Program 2

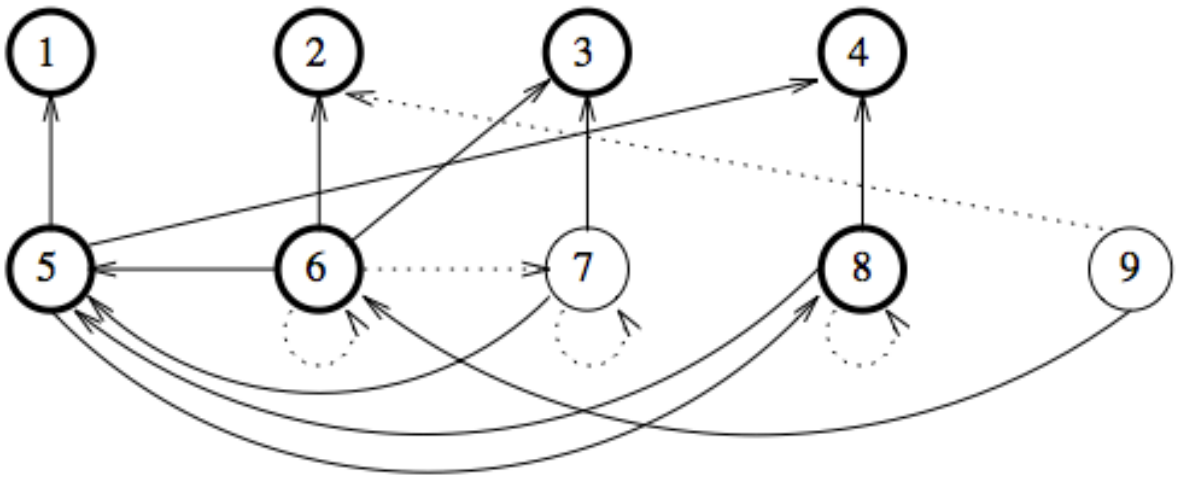


Figure 6: Dynamic Slice using Approach 2 for the program in Figure 3, test-case N = 1, for variable Z, at the end of execution. All edges are drawn as dotted at the beginning. An edge is made solid if the corresponding dependency is ever activated during execution. Only solid edges are traversed while slicing; nodes in the bold denote the slice obtained.

Trace order: <1,2,3,4,5^1,6,7,8,5^2,9>
N=1

Dynamic Slicing 2

- In addition to marking the nodes, mark edges (dependencies) of the graph only as they come up during program execution.
- Find the dynamic slice on this subgraph.

- Problem:
  - If the program contains loops then this method may again overestimate the true dynamic slice.

```
        begin
S1:         read(N);
S2:         I := 1;
S3:         while (I <= N)
            do
S4:             read(X);
S5:             if (X < 0)
                then
S6:                 Y := f1(X);
                else
S7:                 Y := f2(X);
                end_if;
S8:             Z := f3(Y);
S9:             WRITE(Z);
S10:            I := I + 1;
            end_while;
        end.
```

Figure 7: Example Program 3

Trace: <1,2,3^1,4^1,5^1,6,8^1,9^1,10^1,3^2,4^2,5^2,7,8^2,9^2,10^2>
N=2, X= -4 then 3.

```
        begin
S1:         read(N);
S2:         I := 1;
S3:         while (I <= N)
            do
S4:             read(X);
S5:             if (X < 0)
                then
S6:                 Y := f1(X);
                else
S7:                 Y := f2(X);
                end_if;
S8:             Z := f3(Y);
S9:             WRITE(Z);
S10:            I := I + 1;
            end_while;
        end.
```

Figure 7: Example Program 3

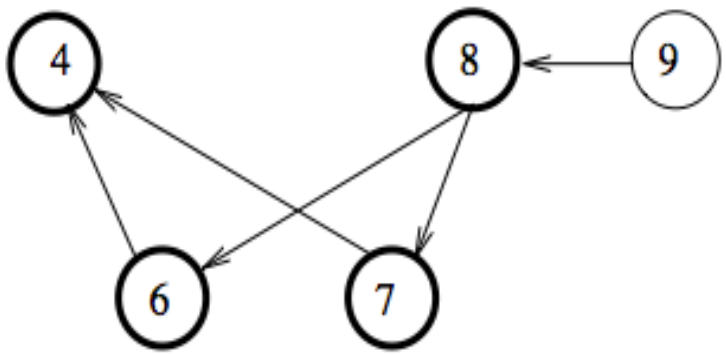


Figure 8: A subset of the dynamic slice obtained using Approach 2 for the program in Figure 7, test-case (N = 2, X = −4, 3), for Variable Z. Node 6 should not be in the slice!

Trace: <1,2,3^1,4^1,5^1,6,8^1,9^1,10^1,3^2,4^2,5^2,7,8^2,9^2,10^2>

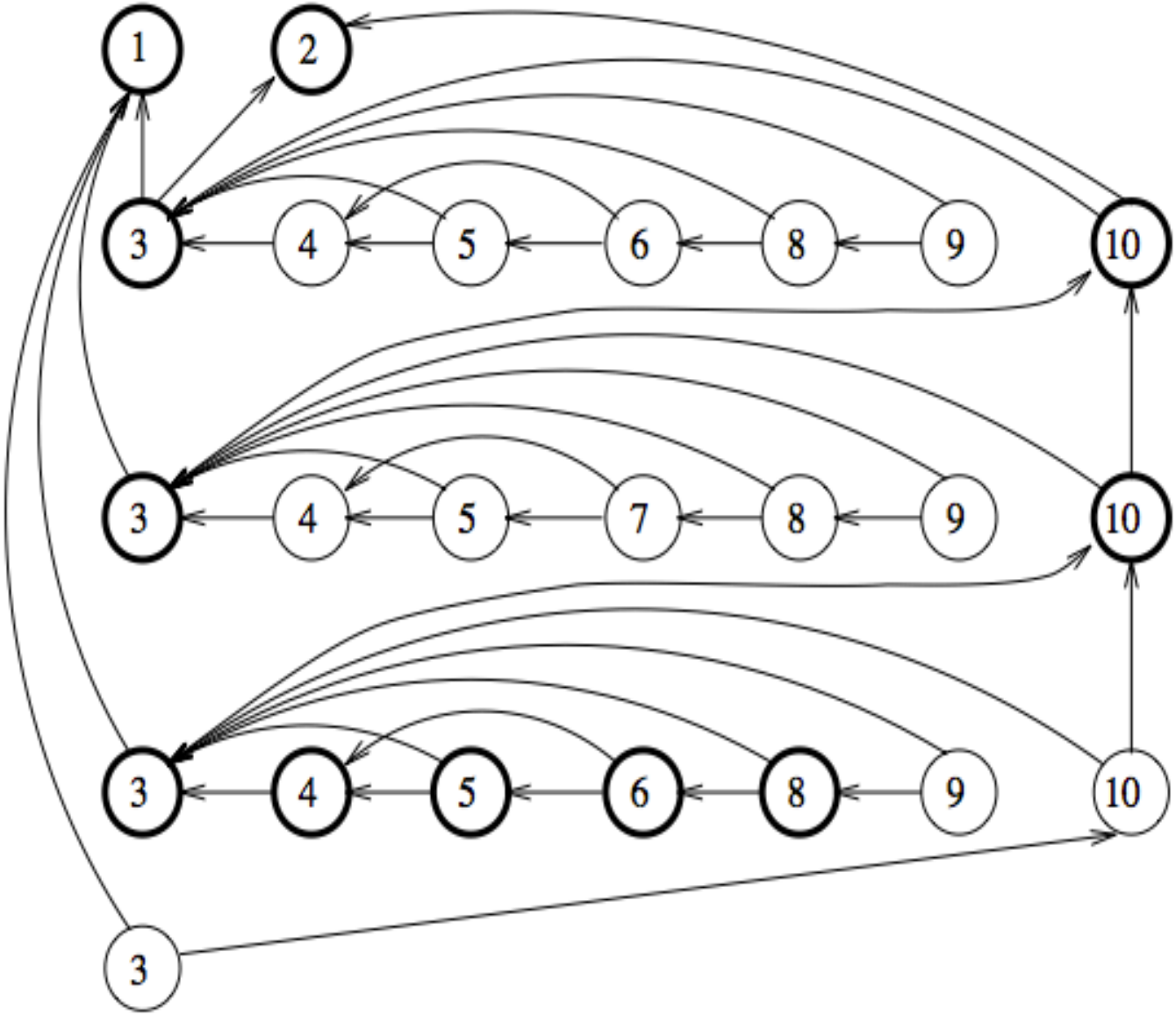
N=2, X= -4 then 3.

Dynamic Slicing 3

- Observation: Each time a statement is executed in the program, it may be dependent on different prior statements.
- Create a separate node every time we encounter a statement in the execution history, and show the dependencies for that specific instance of the statement.

```
        begin
S1:             read(N);
S2:             I := 1;
S3:             while (I <= N)
                do
S4:                 read(X);
S5:                 if (X < 0)
                    then
S6:                     Y := f1(X);
                    else
S7:                     Y := f2(X);
                    end_if;
S8:                 Z := f3(Y);
S9:                 WRITE(Z);
S10:                I := I + 1;
                end_while;
        end.
```

Figure 7: Example Program 3




```
        begin
S1:         read(N);
S2:         I := 1;
S3:         while (I <= N)
            do
S4:             read(X);
S5:             if (X < 0)
                then
S6:                 Y := f1(X);
                else
S7:                 Y := f2(X);
                end_if;
S8:             Z := f3(Y);
S9:             WRITE(Z);
S10:            I := I + 1;
            end_while;
        end.
```

Figure 7: Example Program 3

Figure 10: Dynamic Dependence Graph for the Program in Figure 7 for the test-case (N = 3, X = −4, 3, −2). Nodes in bold give the Dynamic Slice for this test-case with respect to variable Z at the end of execution.

N=2, X= -4 then 3 then -2.

Dynamic Slicing 4

- Clear problem with the prior approach is that the graph will grow with the length of the execution.
- However, as a dynamic slice is a subset of a finite program, there is only a finite set of dynamic slices.
- Try to restrict the graph in approach 3.

Dynamic Slicing 4

- *DefnNode*
  - Maps variable names to the last node that assigned a value to that variable.
- *PredNode*
  - Maps control statements to node corresponding to the its last occurrence in the execution history.
- *ReachableStmts*
  - Each node gets a set of all statements that can be reached from that node.

Dynamic Slicing 4

- Construct Program dependence graph the same manner as in 3, but track all dependencies of the node in the reachable statements set.

- Create a new node only if another node with the same label and same dependencies does not already exist. *DefnNode* and *PredNode* tables tell us what nodes to attach it to.

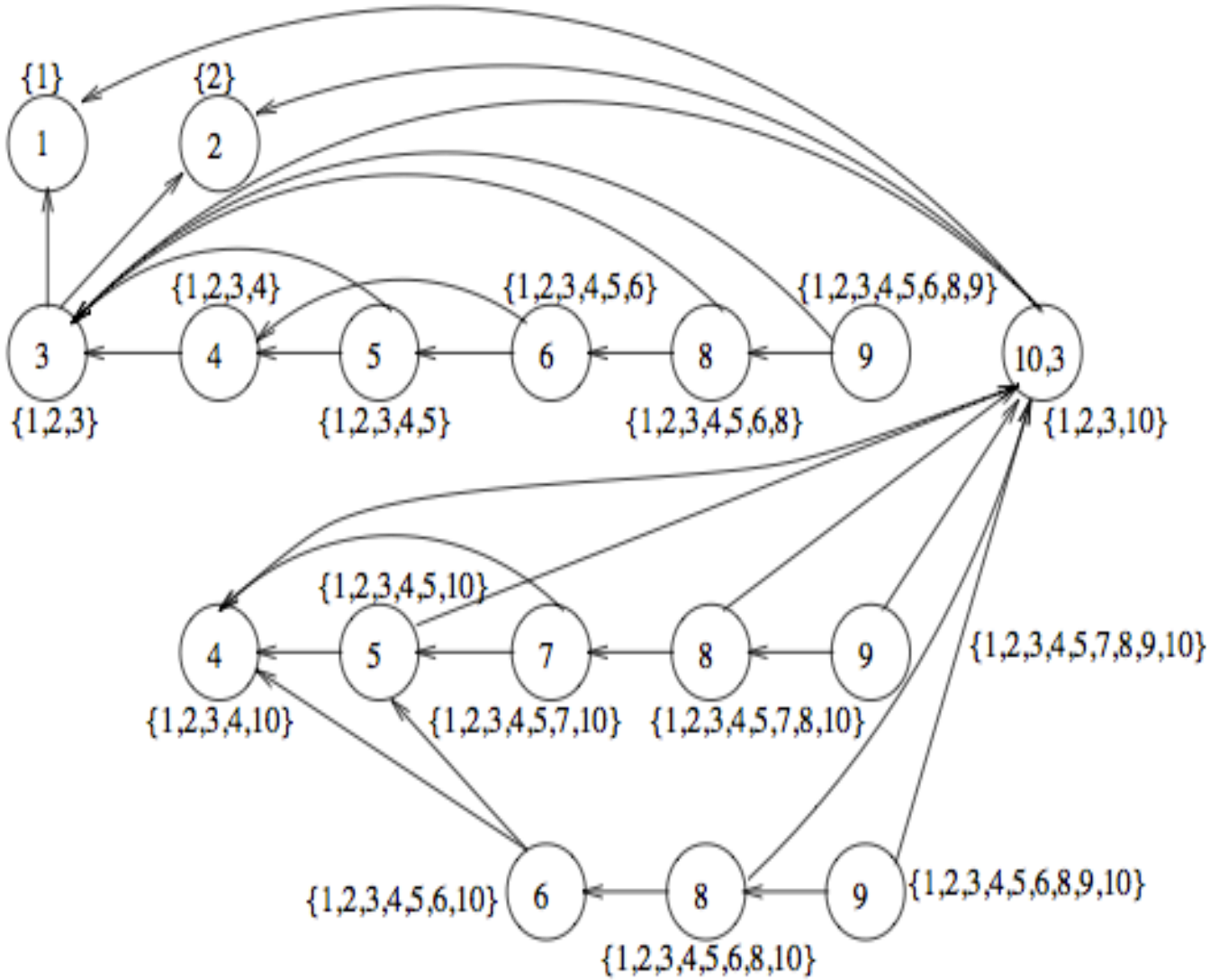
Dynamic Slicing 4

- If a new node has a subset of reachable statements of one of its predecessors (either a control or data dependency), merge it with the predecessor and add any new dependencies.
  - Helps to avoid circular dependencies arising in the graph.

- Use *DefnNode* to find last node that assigned to our slice variable, that node's *ReachableStmts* gives the dynamic slice of the graph.

```
        begin
S1:             read(N);
S2:             I := 1;
S3:             while (I <= N)
                do
S4:                read(X);
S5:                if (X < 0)
                   then
S6:                      Y := f1(X);
                   else
S7:                      Y := f2(X);
                   end_if;
S8:                Z := f3(Y);
S9:                WRITE(Z);
S10:               I := I + 1;
                end_while;
        end.
```

Figure 7: Example Program 3




```
        begin
S1:         read(N);
S2:         I := 1;
S3:         while (I <= N)
            do
S4:             read(X);
S5:             if (X < 0)
                then
S6:                 Y := f1(X);
                else
S7:                 Y := f2(X);
                end_if;
S8:             Z := f3(Y);
S9:             WRITE(Z);
S10:            I := I + 1;
            end_while;
        end.
```

Figure 7: Example Program 3

Figure 11: The Reduced Dynamic Dependence Graph for the Program in Figure 7 for the test-case (N = 3, X = −4, 3, −2), obtained using Approach 4. Each node is annotated with *ReachableStmts*, the set of all statements reachable from that node.

N=2, X= -4 then 3 then -2.