

A New Foundation For Control-Dependence and Slicing for Modern Program Structures

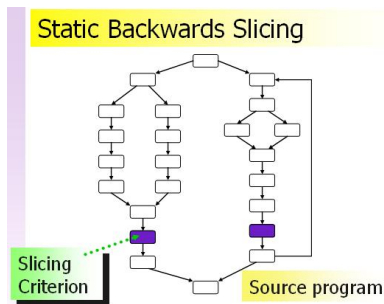
Venkatesh Prasad Ranganath¹ Torben Amtoft¹
Anindya Banerjee¹ Matthew B. Dwyer² John Hatcliff¹

¹Kansas State University

²University of Nebraska, Lincoln

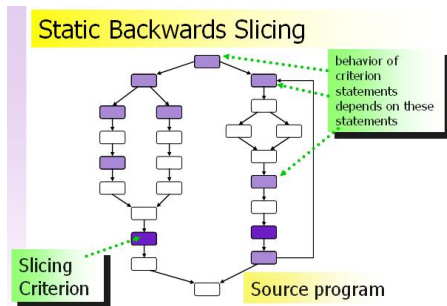
ESOP in Edinburgh, April 6, 2005

What is slicing?



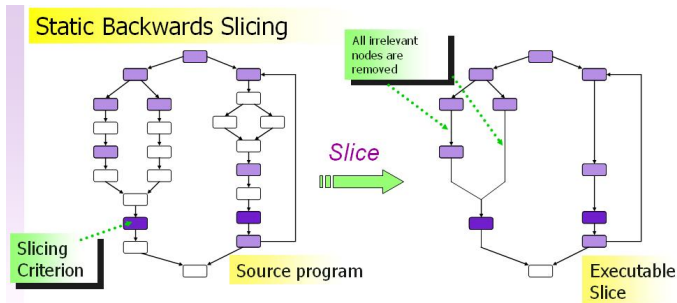
Pick one or more program points of interest
(called the slicing criterion C)

What is slicing?



Walk backwards to find nodes that influence nodes (called the slice set S_C)

What is slicing?



Remove irrelevant nodes

Control & data dependence

A Program Slicer uses *dependence information* to calculate the set of relevant statements

```
public void calculate2() {  
    final Random _r = new Random();  
    int _result1 = _r.nextInt(100);  
  
    int a = 0;  
    int b = 0;  
    int c = 0;  
  
    if (_result1 <= 50) {  
        a = a + 1;  
        b = b + 1;  
        c = c + 1;  
    }  
}
```

Control Dependence

...depends on the conditional statement which controls whether or not the current statement is executed

Data Dependence

...depends on the data value assigned to *c*

Our context

The screenshot displays the Eclipse IDE interface. The main editor shows a Java class with a method `showInterpretedDataFlowThroughTheFilter()`. Several lines of code are highlighted in green, indicating the current execution path. Two dependency analysis views are overlaid on the code:

- Dependence History View:** Shows a list of statements and their dependencies. The current statement is `if ($$ == 0) gates return 0;`, which is data-dependent on `return true;` and `return false;` from the `Control` block.
- Dependence Tracking View:** Shows a table of statements, filenames, line numbers, and their relations to previous items.

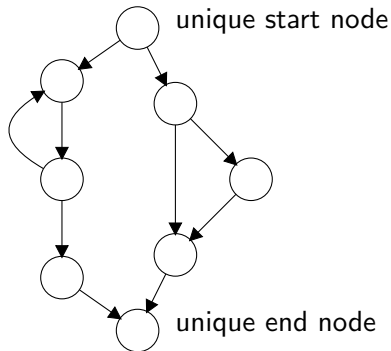
Statement	Filename	Line number	Relation with previous item
<code>objectW = new Object();</code>	ReadersWriters.java	14	Data Dependent
<code>objectW = new Object();</code>	ReadersWriters.java	14	Interference Dependee
<code>synchronized(objectW)</code>	ReadersWriters.java	54	Starting Program Point
<code>synchronized(this)</code>	ReadersWriters.java	45	Synchronization Dependee
<code>nr--;</code>	ReadersWriters.java	47	Interference Dependent
<code>nr++;</code>	ReadersWriters.java	37	Interference Dependee
<code>nr--;</code>	ReadersWriters.java	47	Data Dependee
<code>nr++;</code>	ReadersWriters.java	37	Control Dependent
<code>if(nr == 0)</code>	ReadersWriters.java	35	Starting Program Point

Slicer for Java, being applied to large scale systems

Concerns for our work

Classic definitions of control dependence
assume unique end node

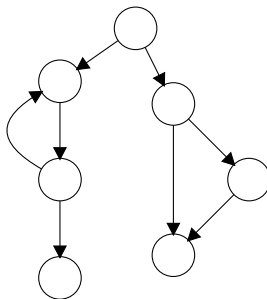
Many Java programs
have CFG's that fail
to satisfy this



Concerns for our work

Multiple end nodes

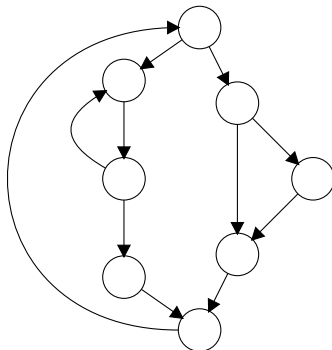
- ▶ multiple return in method body
- ▶ exception returns



Concerns for our work

No end node

- ▶ thread bodies in reactive Java systems
- ▶ “exits” when killed



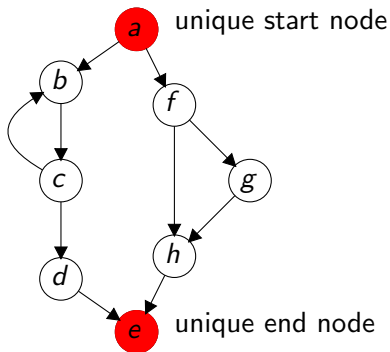
e.g., event handler for GUI thread

Our contribution

This talk **generalizes** classic definitions to handle the cases of no end nodes, or multiple end nodes.

Classic definitions

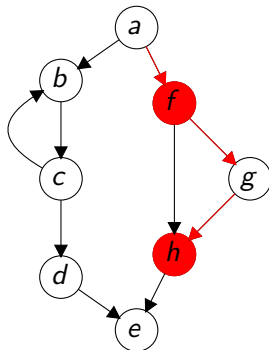
Control Flow Graph



Classic definitions

Domination

f dominates h

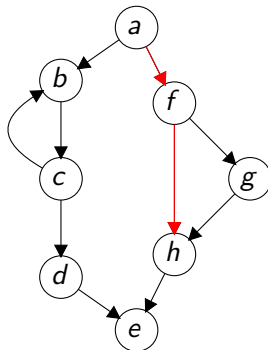


All paths from start node a to h pass through f

Classic definitions

Domination

g does **not** dominate h

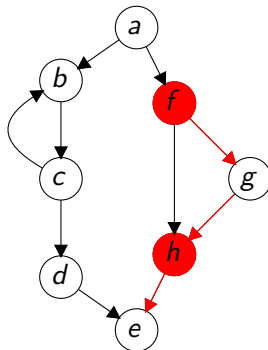


A path from start node a to h not passing through g

Classic definitions

Postdomination

h postdominates f

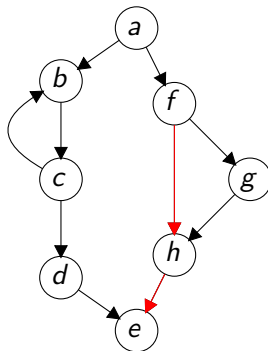


All paths from f to end node e pass through h

Classic definitions

Postdomination

g **not** postdominates f

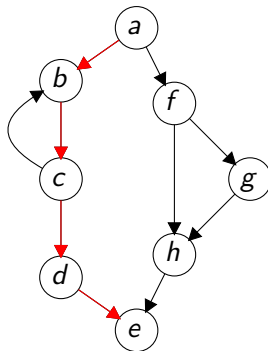


...because there exists a path from f to end node e not passing through g

Classic definitions

Control Dependence

h control dependent on *a*:
a controls whether *h* is
executed or bypassed

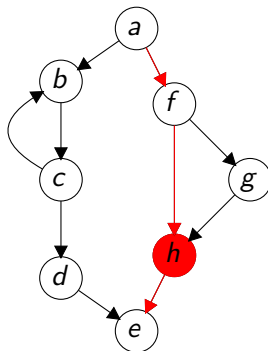


If *a* chooses *b*, *h* can be avoided on path to *e*
(*h* does not postdominate *a*)

Classic definitions

Control Dependence

h control dependent on a :
 a controls whether h is
executed or bypassed

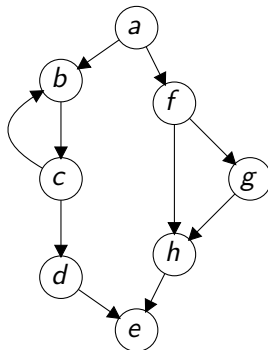


If a chooses f , it commits to h
 h does postdominate f

Classic definitions

Control Dependence

- ▶ g **not** control dep on a
- ▶ but if g in slice set
also a in slice set:
 - ▶ g control dep on f
 - ▶ f control dep on a

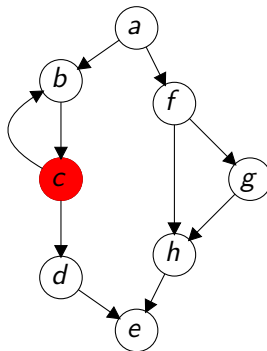


Slicing takes transitive closure of (control) dependence

Classic definitions

Loops: do guards control aftermath?

d not control dep on *c*



c cannot bypass *d* on path to *e*

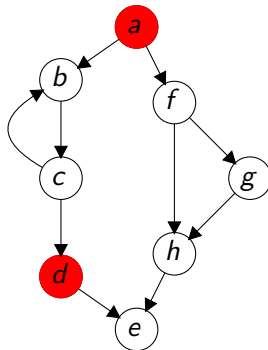
Classic definitions

Loops: do guards control aftermath? **no**

d **not** control dep on *c*

Slicing criterion: *d*

yields slice set: *a*, *d*

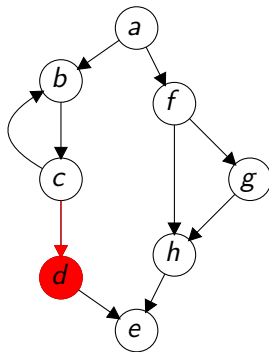


The sliced program may terminate **more** often than the original

Classic definitions

Less well-known definition (Podgurski & Clarke),
sensitive to non-termination

d weakly control dep on c

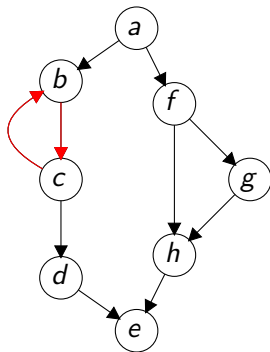


If c selects d , it commits to d

Classic definitions

Less well-known definition (Podgurski & Clarke),
sensitive to non-termination

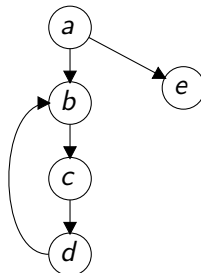
d weakly control dep on c



If c selects b , it can avoid d by looping!

Assessment of classic method

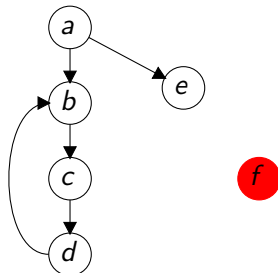
Handling reactive systems by
converting to unique end node



Assessment of classic method

Handling reactive systems by
converting to unique end node

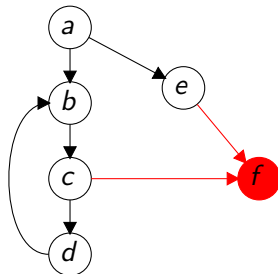
new node f



Assessment of classic method

Handling reactive systems by
converting to unique end node

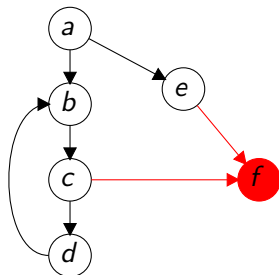
make f end node
Choice of edges ad hoc



Assessment of classic method

Handling reactive systems by
converting to unique end node

Choice of edges ad hoc
Dependencies changed



b now control dependent on *c*
d no longer control dependent on *a*

Key insight

We propose a new definition based on the following idea.

When is h control dependent on a ?

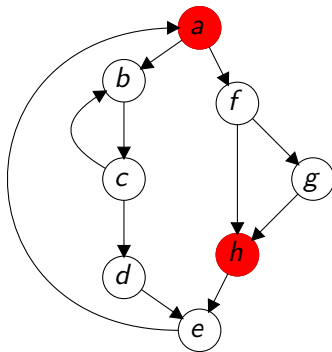
- ▶ from one of a 's successors, h **cannot** be avoided forever:
all **maximal** paths go through h .
- ▶ from another of a 's successors, h **may** be avoided forever:
there exists a maximal path not going through h .

Note that **no** mention of “end node”!

Control dependence based on Maximal paths

Illustrating example

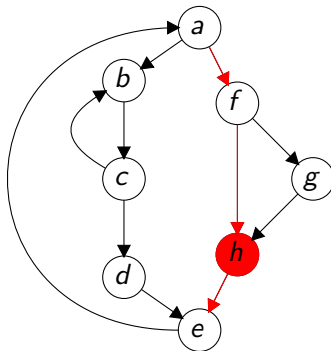
h control dependent on *a*



Control dependence based on Maximal paths

Illustrating example

h control dependent on *a*

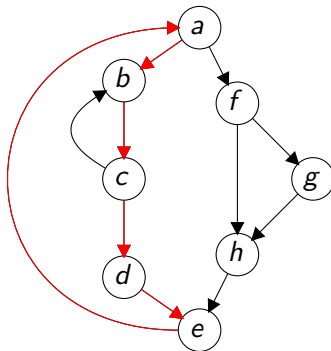


If *a* selects *f* then *h* cannot be avoided

Control dependence based on Maximal paths

Illustrating example

h control dependent on *a*

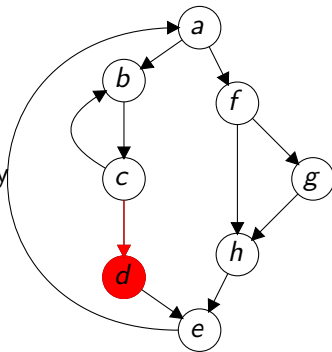


If *a* selects *b* then *h* can be avoided forever

Control dependence based on Maximal paths

Loop guards control aftermath

d control dependent on c
so inner loop **not** sliced away

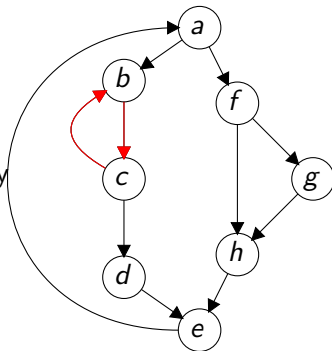


If c selects d then d cannot be avoided

Control dependence based on Maximal paths

Loop guards control aftermath

d control dependent on *c*
so inner loop **not** sliced away



If *c* selects *b* then *d* can be avoided forever

Non-Termination Sensitive Control Dependence

Our New Definition

In a CFG, b is NTSCD on a iff

- ▶ a has two successors c and d ;
- ▶ on all maximal paths from c , b occurs;
- ▶ there exists a maximal path from d on which b does not occur

We use this definition in our slicer

Non-Termination Sensitive Control Dependence

Conservative Extension

In the **special case** where the CFG has unique end node, NTSCD is equivalent to Podgurski & Clarke's "weak control dependence"

Assessment

- ▶ New CD definition for CFG without unique end restriction

Assessment

- ▶ New CD definition for CFG without unique end restriction
- ▶ Sensitive to (preserves) non-termination

Assessment

- ▶ New CD definition for CFG without unique end restriction
- ▶ Sensitive to (preserves) non-termination
- ▶ Slices are typically larger due to this stronger notion:
 - ▶ great, if you are slicing to preserve liveness properties for model checking
 - ▶ not so great, if slicing for program understanding

Assessment

- ▶ New CD definition for CFG without unique end restriction
- ▶ Sensitive to (preserves) non-termination
- ▶ Slices are typically larger due to this stronger notion:
 - ▶ great, if you are slicing to preserve liveness properties for model checking
 - ▶ not so great, if slicing for program understanding
- ▶ So can we generalize the termination **in**sensitive definition that most people use?

Non-Termination Insensitive Control Dependence

Key idea: generalize “end node” to “control sink”

- ▶ an end node is one node where control ends
- ▶ a control sink is a section of graph which if entered is never exited

Definition: In a CFG, b is NTICD on a iff

- ▶ a has two successors c and d ;
- ▶ on all **sink-bounded** paths from c , b occurs;
- ▶ there exists a **sink-bounded** path from d on which b does not occur

Non-Termination Insensitive Control Dependence

Key idea: generalize “end node” to “control sink”

- ▶ an end node is one node where control ends
- ▶ a control sink is a section of graph which if entered is never exited

Definition: In a CFG, b is NTICD on a iff

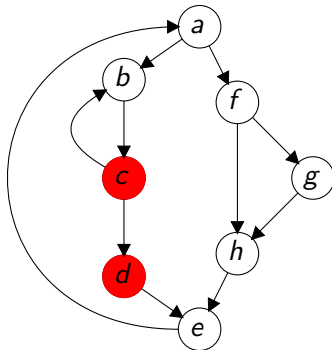
- ▶ a has two successors c and d ;
- ▶ on all **sink-bounded** paths from c , b occurs;
- ▶ there exists a **sink-bounded** path from d on which b does not occur

Theorem: In the **special case** where unique end node, this is equivalent to classical control dependence.

Control dependence based on Sink-bounded paths

Example: loop guard does not control aftermath

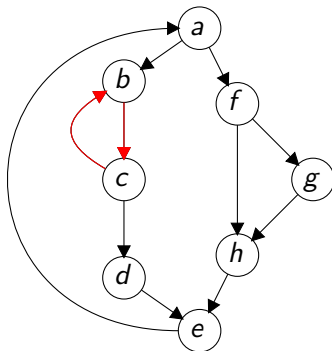
d **not** NTICD on c



Control dependence based on Sink-bounded paths

Example: loop guard does not control aftermath

d **not** NTICD on *c*

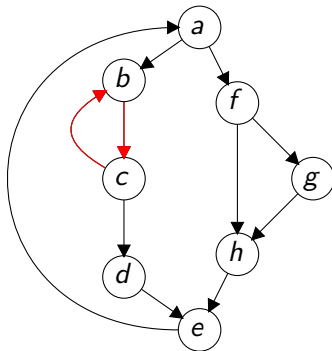


Path avoids *d*, but is not sink-bounded

Control dependence based on Sink-bounded paths

Example: loop guard does not control aftermath

b,c **not** control sink

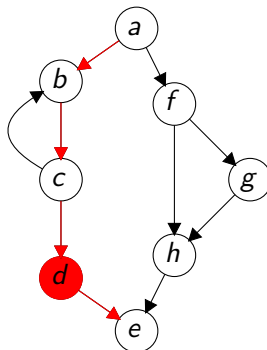


Strongly connected, but outgoing edges

NTSCD generates a larger slice set than NTICD

NTSCD itself may be smaller

▶ d NTICD on a

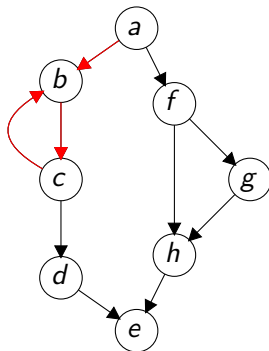


If a selects b , any path to sink e goes through d

NTSCD generates a larger slice set than NTICD

NTSCD itself may be smaller

- ▶ d NTICD on a
- ▶ d **not** NTSCD on a

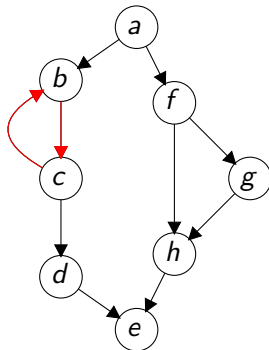


If a selects b , d can still be avoided

NTSCD generates a larger slice set than NTICD

But the closure of NTSCD is larger

- ▶ d NTICD on a
- ▶ d **not** NTSCD on a
- ▶ d NTSCD on c

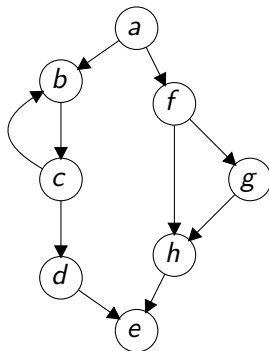


If c selects b , d can still be avoided

NTSCD generates a larger slice set than NTICD

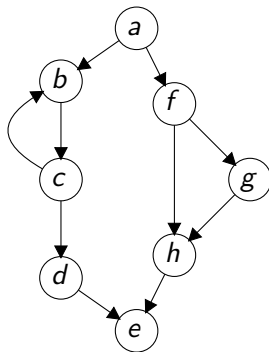
But the closure of NTSCD is larger

- ▶ d NTICD on a
- ▶ d **not** NTSCD on a
- ▶ d NTSCD on c
- ▶ c NTSCD on a



If a selects b , c cannot be avoided

NTSCD generates a larger slice set than NTICD



Trade-off: small slice vs. maintain liveness properties

Control Dependence in Computation Tree Logic

We have debugged our definitions by dumping a bunch of CFGs into a model checker and automatically checking them against the formulae below:

b is NTSCD on a iff

$$(G, a) \models \text{EX}(\text{AF}(b)) \wedge \text{EX}(\text{EG}(\neg b))$$

- ▶ from one of a 's successors, b **cannot** be avoided forever:
all maximal paths contain b .
- ▶ from another of a 's successors, b **may** be avoided forever:
there **exists** a maximal path not containing b .

Control Dependence in Computation Tree Logic

We have debugged our definitions by dumping a bunch of CFGs into a model checker and automatically checking them against the formulae below:

b is NTICD on a iff (when restricted to sink-bounded paths)

$$(G, a) \models \text{EX}(\text{AF}(b)) \wedge \text{EX}(\text{EG}(\neg b))$$

- ▶ from one of a 's successors, **all** sink-bounded paths contain b .
- ▶ from another of a 's successors, there **exists** a sink-bounded path not containing b .

Control Dependence in Computation Tree Logic

We have debugged our definitions by dumping a bunch of CFGs into a model checker and automatically checking them against the formulae below:

b is NTSCD on a iff

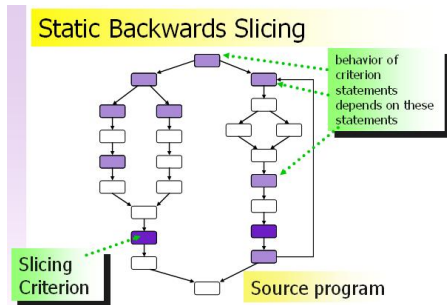
$$(G, a) \models \text{EX}(\text{AF}(b)) \wedge \text{EX}(\text{EG}(\neg b))$$

General setting

With **arbitrary execution traces** we define that b is NTSCD on a iff

$$(G, a) \models \text{EX}(\text{A}[\neg a \text{U} b]) \wedge \text{EX}(\text{E}[\neg b \text{W}(\neg b \wedge a)]).$$

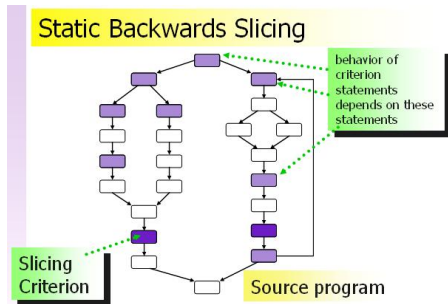
Slicing Transformation (conceptually)



Assumptions: Slice set S_C backwards closed under

- ▶ NTSCD
- ▶ and data dependency

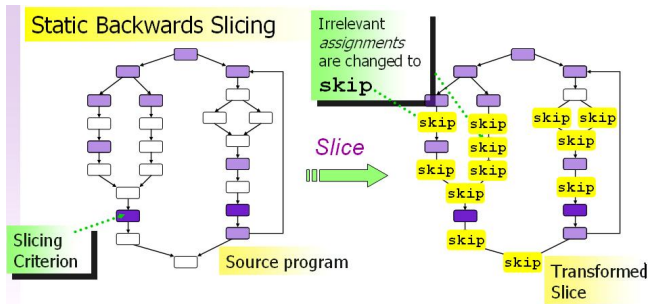
Slicing Transformation (conceptually)



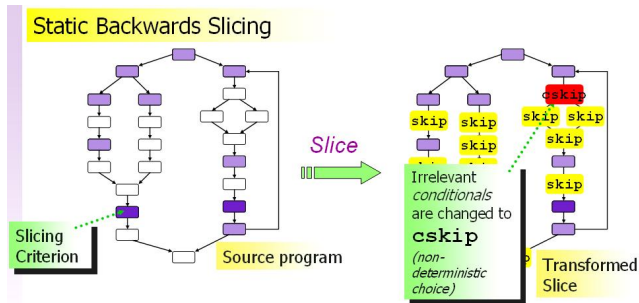
Assumptions: Control graph **reducible**:

- ▶ forward edges form a DAG;
- ▶ back edges (target dominates source)

Slicing Transformation (conceptually)

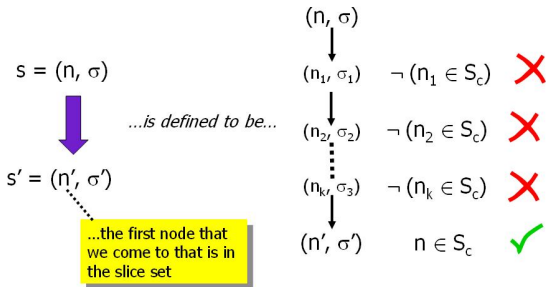


Slicing Transformation (conceptually)



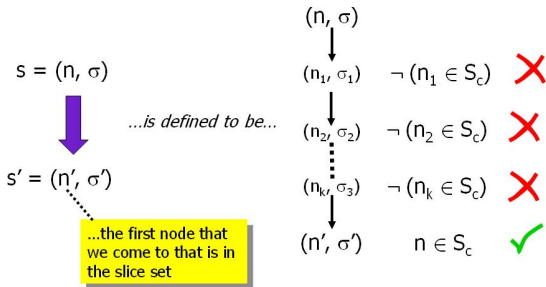
Bisimulation-based Correctness Result

Reduction until observable



Bisimulation-based Correctness Result

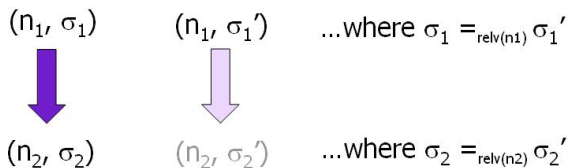
Equality on observable



(n, σ_1) and (n, σ_2) are **equal on observables**
 provided σ_1 and σ_2 agree on $relv(n)$,
 the variables not redefined before they are used by an observable

Bisimulation-based Correctness Result

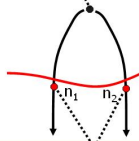
Equality on observable is a Bisimulation



Property of Unique First Observable

Key property of correctness proof

A conditional that
is not in the slice...



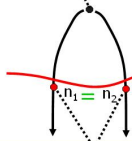
Boundary representing
nodes *in* the slice

First node in slice
encountered along path of
non-relevant nodes

Property of Unique First Observable

Key property of correctness proof

*A conditional that
is not in the slice...*

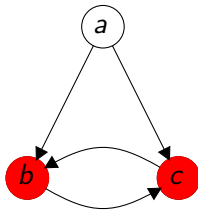


Boundary representing
nodes *in* the slice

*First node in slice
encountered along path of
non-relevant nodes*

Need for Reducible CFG

Slice set is $\{b, c\}$



as neither b nor c is NTSCD on a

Algorithm

- ▶ Algorithms have been written

Algorithm

- ▶ Algorithms have been written
- ▶ and implemented for use in a slicer

Algorithm

- ▶ Algorithms have been written
- ▶ and implemented for use in a slicer
- ▶ described in FASE talk Friday!

Future work

- ▶ Extend to irreducible CFG
- ▶ Extend to general execution traces
(not necessarily from CFG)

Conclusion

Slicing for modern control structures: Our definitions

- ▶ have sound semantic foundation
- ▶ even preserve termination
- ▶ can be implemented to handle 10K+ lines of code and still return results in seconds (calculation of control dependence not bottleneck)