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

Venkatesh Prasad Ranganath\textsuperscript{1} Torben Amtoft\textsuperscript{1}
Anindya Banerjee\textsuperscript{1} Matthew B. Dwyer\textsuperscript{2} John Hatcliff\textsuperscript{1}

\textsuperscript{1}Kansas State University

\textsuperscript{2}University 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.

```java
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

Slicer for Java, being applied to large scale systems

Ranganath, Amtoft, Banerjee, Dwyer, Hatcliff

New Foundation for Control-Dependence and Slicing
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
This talk *generalizes* classic definitions to handle the cases of no end nodes, or multiple end nodes.
Classic definitions

Control Flow Graph

- unique start node
- unique end node
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 $h$
Postdomination

\( g \) not postdominates \( f \)

...because there exists a path from \( f \) to end node \( e \) not passing through \( g \)
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 $a$)
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?

\[ d \text{ not control dep on } c \]

\[ c \text{ cannot bypass } d \text{ on path to } e \]
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

\[ d \text{ weakly control dep on } c \]

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

Ranganath, Amtoft, Banerjee, Dwyer, Hatcliff New Foundation for Control-Dependence and Slicing
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 $a$
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

If a selects b then h can be avoided forever
Control dependence based on Maximal paths

Loop guards control aftermath

If $c$ selects $d$ then $d$ cannot be avoided

$d$ control dependent on $c$
so inner loop not sliced away
Control dependence based on Maximal paths

Loop guards control aftermath

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
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 insensitive 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

\[ d \text{ not NTICD on } c \]

Path avoids \( d \), but is not sink-bounded
Control dependence based on Sink-bounded paths

Example: loop guard does not control aftermath

\[ b, c \text{ not control sink} \]

Strongly connected, but outgoing edges
NTSCD generates a larger slice set than NTICD

NTSCD itself may be smaller

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

\[ (G, a) \models EX(AF(b)) \land EX(EG(\neg b)) \]

- 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:

\[ (G, a) \models EX(AF(b)) \land EX(EG(\neg b)) \]

- 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

$$(G, a) \models EX(AF(b)) \land EX(EG(\neg b))$$

**General setting**

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

$$(G, a) \models EX(A[\neg aUb]) \land EX(E[\neg bW(\neg b \land a)])$$
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)

Static Backwards Slicing

Irrelevant assignments are changed to skip

Slice

Transformed Slice

Slicing Criterion

Source program

Ranganath, Amtoft, Banerjee, Dwyer, Hatcliff

New Foundation for Control-Dependence and Slicing
Slicing Transformation (conceptually)

Static Backwards Slicing

Slicing Criterion

Source program

Slice

Irrelevant conditionals are changed to
\texttt{cskip}

(non-deterministic choice)

Transformed Slice

Ranganath, Amtoft, Banerjee, Dwyer, Hatcliff

New Foundation for Control-Dependence and Slicing
Bisimulation-based Correctness Result

Reduction until observable

s = (n, σ)

...is defined to be...

s' = (n', σ')

...the first node that we come to that is in the slice set

(n, σ)

(n_1, σ_1) \neg (n_1 \in S_c) \times

(n_2, σ_2) \neg (n_2 \in S_c) \times

(n_k, σ_3) \neg (n_k \in S_c) \times

(n', σ') n \in S_c \checkmark
Bisimulation-based Correctness Result

Equality on observable

\[(n, \sigma_1) \quad \text{and} \quad (n, \sigma_2) \text{ are equal on observables provided } \sigma_1 \text{ and } \sigma_2 \text{ agree on } relv(n), \text{ the variables not redefined before they are used by an observable.} \]
Bisimulation-based Correctness Result

Equality on observable is a Bisimulation

\[(n_1, \sigma_1) \quad (n_1, \sigma_1') \quad \ldots \text{where } \sigma_1 =_{\text{rel}(n_1)} \sigma_1'\]

\[(n_2, \sigma_2) \quad (n_2, \sigma_2') \quad \ldots \text{where } \sigma_2 =_{\text{rel}(n_2)} \sigma_2'\]
Bisimulation-based Correctness Result

Equality on observable is a Bisimulation

\[(n_1, \sigma_1) \quad (n_1, \sigma'_1) \quad \ldots \text{where } \sigma_1 = _{\text{relv}(n_1)} \sigma'_1\]

\[(n_2, \sigma_2) \quad (n_2, \sigma'_2) \quad \ldots \text{where } \sigma_2 = _{\text{relv}(n_2)} \sigma'_2\]
Property of Unique First Observable

Key property of correctness proof

A conditional that is not in the slice...

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

Boundary representing nodes in the slice
Property of Unique First Observable

Key property of correctness proof

A conditional that is not in the slice...

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

Boundary representing nodes in the slice
Need for Reducible CFG

Slice set is \{b, c\}

as neither b nor c is NTSCD on a
Algorithms have been written
Algorithm

- Algorithms have been written
- and implemented for use in a slicer
Algorithms have been written
and implemented for use in a slicer
described in FASE talk Friday!
Motivation
Classical Perspective
New Perspective
Application for Slicing
Conclusion

Screenshots for slicer
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)