Program Dependencies
Reading assignment

Today's reading


• Background
Program Slice

- Introduced by Mark Weiser in 1979
- Argued it was a mental abstraction that programmers used when debugging
- Program slice $S$ is a reduced, executable program obtained from program $P$ by removing statements from $P$, such that $S$ replicates part of the behavior of $P$
read n;
i := 1;
sum := 0;
product := 1;
while i ≤ n do
    sum := sum + 1;
    product := product * i;
i := i + 1;
endwhile;
write sum;
write product;
Original Slicing Concept

- Based on statements
- Algorithm restricted to structured programs
- Missed some relationships
- Foundation for considerable work
  - Podgurski and Clarke generalized some of the concepts
    - Language-independent model of dependence
    - More general model of control flow
      - Weak and strong control flow
Applications of Program Slicing/Dependence

• Debugging
• Data/control flow testing criteria
• Software understanding
• Maintenance
Program Dependencies

- $s_k$ is semantically dependent on $s_i$ if the semantics of $s_i$ can affect the execution behavior of $s_k$

- In general, can’t determine semantic dependence
**Syntactic Dependence**

- Can we find **syntactic dependence** relations that “approximate” semantic dependence?
  - that define necessary conditions for semantic dependence
  - that are defined in terms of a language-independent, graph-theoretic model that can be efficiently computed
Forward Dominators

- let $G(N,E)$ be a control flow graph, where $s_u$, $s_v$, and $s_f$ are nodes in $G$ and $s_f$ is the final node
  - a node $s_v$ forward dominates $s_u$ iff every $s_u \rightarrow s_f$ path in $G$ contains $s_v$
  - $s_v$ properly forward dominates $s_u$ iff
    - $s_u \neq s_v$ and $s_v$ forward dominates $s_u$
Immediate Forward Dominators

A node $s_u$ is the immediate forward dominator of $s_u$, $s_u \neq s_f$ if it is the node that is the first proper forward dominator of $s_u$ to occur on every $s_u \rightarrow s_f$ path in $G$ after a branch, this is the point where all paths come together.
Example

7 forward dominates all nodes

$\text{ifd}(5) = 7$
$\text{ifd}(1) = 4$
$\text{ifd}(4) = 5$
Control Dependence

• $s_v$ is control dependent on $s_u$ iff there exists a path $s_u \cdot P \cdot s_v$ not containing the immediate forward dominator of $s_u$
  • bodies of structured constructs are control dependent on the start of the construct
Example

- 2 and 3 are control dependent on 1
- 6 is control dependent on 5
**Direct Data Dependence**

Assume $G(N,E)$ is a control flow graph, where $\text{Def}(s_n)$ are the variables defined at node $s_n$ and $\text{Use}(s_n)$ are the variables referenced at node $s_n$.

If path $P = s_{i1}, \ldots, s_{in}$ then $\text{Def}(P) = \bigcup_j \text{Def}(s_{ij})$

Node $s_v$ is *directly data dependent* on node $s_u$ iff there is a path $s_u \circ P \circ s_v$ such that

$$(\text{Def}(s_u) \cap \text{Use}(s_u)) - \text{Def}(P) \neq 0$$
In other words

\[(\text{Def}(s_u) \cap \text{Use}(s_u)) - \text{Def}(P) \neq 0\]

This is just a set theoretic way of saying that there is at least one variable, say \(x\), defined at node \(s_u\) that is used at node \(s_v\) and there is a def-clear path with respect to \(x\) from \(s_u\) to \(s_v\).
Data Dependence

- node $s_u$ is data dependent on $s_u$ iff there is a path $s_{v1},...,s_{vn}$ such that $u=v1$, $v=vn$, and $S_{vi}$ is directly data dependent on $S_{vi+1}$ for all $i$, $1 \leq i < n$.
Example

- 4 is **directly data dep.** on 2
- 6 is **directly data dep.** on 4
- 6 is **data dependent** on 2

**directly data dependent** is the same as the def-use relationship defined by Rapps and Weyuker

**data dependent** is the same as the chains of def-ref used by Ntafos, but without a bound
Syntactic Dependence

- node $S_v$ is syntactically dependent on $S_u$ iff there is a path $S_{v_1}, \ldots, S_{v_n}$ of nodes such that $u=v_1$, $v=v_n$, and $S_{v_{i+1}}$ is data or control dependent on $S_{v_i}$ for all $i$, $1 \leq i < n$
  
  - combines data dependence and control dependence

- sometimes called information flow

- syntactic dependence over-approximates semantic dependence
  
  - Why?
Syntactic Dependence

Control flow
(1,2)
(1,3)
(5,7)
(5,6)

Direct Data flow
(2,7)
(6,7)
Data (and control) flow coverage criteria

- coverage criteria exercise subsets of control and data dependencies in the hope of exposing faults
- Rapps and Weyuker, Ntafos, Laski and Korel selected different subsets of information flow
- need experimental data to know which are the most effective subsets
  - intuitively, direct data dependence and control dependence are appealing
  - relatively easy to achieve at least 85% coverage with automated support
Data Flow/Control Flow Coverage revisited

“NEW Winner”

ORDERED CONTEXT COVERAGE

CONTEXT COVERAGE

REACH COVERAGE

All-DU-Paths

Required \( k \)-Tuples

All-Uses

All-C-Uses/Some-P-Uses

All-P-Uses/Some-C-Uses

All-Defs

All-P-Uses

All-Edges

All-Nodes

All-dependingencies
Applications

- Debugging
- Data/control flow testing criteria
- Software understanding
- Maintenance
Symmetric Relationship

- $\text{dep}(s_i, s_j)$ is true if $s_j$ is syntactically dependent on $s_i$

- $\text{dep}(?, s_j) = \{s_i \mid \text{dep}(s_i, s_j)\}$ is the set of nodes that can syntactically affect $s_j$

- $\text{dep}(s_i, ?) = \{s_j \mid \text{dep}(s_i, s_j)\}$ is the set of nodes that can be syntactically affected by $s_i$
Debugging Dependencies

• Which statements could have caused an observed failure?

• If $s_v$ computes an erroneous value, want to know the statements $s_v$ is dependent upon?

• $\text{dep}(?, S_v)$
Maintenance Dependencies

• Which statements will be affected by a change?
  - \( \text{which statements are dependent upon } s_u \)
    \[ \text{dep}(s_u, ?) \]

• Will a particular statement be affected by a change?
  • Is there a dependency between \( S_u \) and \( S_v \)?
    \[ \text{dep}(S_u, S_v) \]

• Which statements could affect “this” statement?
  • Which statements are statement \( S_v \) dependent on?
    \[ \text{dep}(?, S_v) \]
Kinds of flow

• Forward flow
  • \text{dep}(S_u, ?)

• Backward flow
  • \text{dep}(?, S_v)
Program Dependence Graph

- Originally proposed by Ottenstein and Ottenstein (Ott), 1984
- Nodes correspond to statements
- Edges correspond to data or control dependencies
- A slice corresponds to all nodes that are reachable from a selected node (forward slice)
read n;
i := 1;
sum := 0;
product := 1;
while i ≤ n do
    sum := sum +1;
    product := product * i;
i := i+1;
endwhile;
write sum;
write product;
read n;
i := 1;
sum := 0;
product := 1;
while i ≤ n do
    sum := sum +1;
    product := product * i;
i := i+1;
endwhile;
write sum;
write product;
read n;
i := 1;
sum := 0;
product := 1;
while i ≤ n do
    sum := sum + 1;
    product := product * i;
i := i + 1;
endwhile;
write sum;
write product;
read n
i := 1
prod := 1
sum := 0
while i ≤ n	do
	sum := sum +1;
	prod := prod * i;
	ehatwhile;
write sum;
write prod;

read n;
i := 1;
sum := 0;
product := 1;
while i ≤ n
do
	sum := sum +1;
	product := product * i;
i := i+1;
endwhile;
write sum;
write product;
Problems with dependence analysis/slicing

- In practice, a program slice is often too big to be useful
- Infeasible paths lead to imprecision
- Complex data structures lead to imprecision
  
  \[
  A[i] := \ldots
  \]

  \[
  B[j] := A[k]
  \]

- Need to use an efficient, interprocedural algorithm
Refining program dependencies

- Dependence/slice wrt a criteria
- Dynamic dependence/slice
Levels of Granularity

- by statement
- by entity
  - e.g., $x := y + z$
  - only look at dependencies on $z$
  - only look at data dependencies on $z$
  - only look at direct data dependencies on $z$
- by component
  - e.g., TASC avionics maintenance system
Dynamic Slice

• First proposed by Laski and Korel, 1988
• Only provides those dependencies that were exercised during a particular execution
• Could also be further refined according to some criteria
  • E.g., dynamic slice and depends on statement n
Conclusion

• program dependencies provide a theory for restricting/focusing attention
  • can allow users to select and refine focus of attention
  • can support different levels of granularity
• Can be used for software understanding, regression testing, debugging, maintenance, and data/control test coverage criteria
Conclusion

- for selecting test cases
  - syntactic dependence alone is not adequate
    - the number of syntactic dependencies in a program can be quadratic in the number of statements
    - a given syntactic dependence may be demonstrated by (infinitely) many paths
    - propagation of a fault through a particular path may depend on the selection of input data
      \[ \Rightarrow \text{must use semantic information} \]