A Static Measure of a Subset of Intra-procedural Data Flow Testing Coverage Based on Node Coverage

E. Merlo¹, G. Antoniol²
¹ École Polytechnique de Montréal,
² IRST-Istituto per la Ricerca Scientifica e Tecnologica

ABSTRACT
In the past years, a number of research works, which have been mostly based on pre and post dominator analysis, have been presented about finding subsets of nodes and edges (called “unrestricted subsets”) such that their traversal during execution (if feasible) exercises respectively all feasible nodes and edges in a Control Flow Graph (CFG).

This paper presents an approach to statically measure a subset of intra-procedural data flow (“all uses”) coverage obtained by exercising an “unrestricted subset” of nodes during testing. This measure indicates the possible degree of data flow testing obtainable while using a weaker test coverage criteria.

The approach has been implemented in C++ on a PC under Linux and results obtained from the analysis of “Gnu find tool”, which is about 16 KLOC of C-language source code, are presented together with discussions and conclusions.

1 Introduction
The goal of software testing is to detect bugs in a program under test, in order to remove defects and increase software quality. This activity consumes at least half of the labor expended to produce a working program [4], thus requiring an enormous amount of resources of all kind (human resources, time and money) for this quality to be achieved.

Two major testing methods are available during the testing phase and are termed functional testing (black box testing) and structural testing (white box testing).

In functional testing, only the functionality of a software component is tested; during this process, the tested component is executed with given input data and the produced output is verified for conformity according to an oracle (the program’s specifications for example), which specifies the expected outcome of a set of tests. Functional testing thus analyzes a software component with the user’s point of view, being interested in what the program does rather than in how it does it. This second concern is the focus point for structural testing, which relies on the internal structure of the tested software component, e.g. on its implementation details, in order to proceed with testing. Structural testing aims at producing input data which will enable the coverage of a given testing criterion, data that will afterwards be used as inputs during a functional testing phase. Ultimately, structural testing should produce input data that will ensure execution of all program paths. This coverage criterion is impossible to attain, due to loop termination problems [4] and infeasible paths. Structural testing criteria are therefore softened, given goals such as program statement coverage, program branch coverage, or data-flow criteria such as definition-use coverage [14].

The goal of node coverage testing is to exercise every node in the program at least once. In [1, 5, 6], techniques are presented that reduce the number of program nodes to be covered while still potentially attaining full node
coverage. These techniques are based on concepts of implication between nodes in the program; intuitively, if all possible executions of the program passing through node \( v_2 \) also pass through node \( v_1 \), we can say that an input data case covering node \( v_2 \) also covers node \( v_1 \). Unconstrained nodes, are therefore identified within a program, with the particularity that coverage of these nodes leads to full node coverage. Note that the terminology used in this article is taken from [5].

Other more sophisticated and powerful testing criteria are branch coverage and data flow coverage [14].

In this paper, a study on the intra-procedural data flow testing coverage obtainable while covering the program nodes using the “unconstrained nodes” testing selection criterion is presented. The objective is to study the data flow coverage power of statements in a program to assess the degree of benefits from a higher level testing criteria such as data flow unit testing, practically achievable while using a lower level one such as unit node coverage. Original definitions of intra-procedural node to define isemi-coverage are presented together with preliminary experimental results obtained from the analysis of about 16 KLOC of C-language source code.

In section 2, Flow Graphs, dominators, post-dominators, and the computation of unconstrained nodes are introduced; section 3 presents the original contribution of this paper, i.e. all the definitions necessary to compute possible and definite data flow semi-coverage based on statement coverage; section 4 describes the environment used to analyze the source code used in the experiments which are presented in section 5; section 6 proposes conclusions and further research directions.

2 Theoretical Concepts

This section presents an overview of the concepts involved in unconstrained nodes \(( UN )\) computation [1, 5, 6].

Program “Word Count” [8], which is presented in figure 1, has been selected as an example.

```c
void main(void)
{
    int inword, nl, nw, nc, c ;
    1 : inword = 0 ;
    2 : nl = 0 ;
    3 : nw = 0 ;
    4 : nc = 0 ;
    5 : c = getc(stdin) ;
    6 : while(c != EOF) {
         7 : nc = nc + 1 ;
         8 : if (c == '\n')
             9 :      nl = nl + 1 ;
         10 : if (c=='\n' || c=='\t' || c==' ') 
            11 :      inword = 0 ;
         12 : else if (inword == 0) {
             13 :      inword = 1 ;
             14 :      nw = nw + 1 ;
            }
         15 : c = getc(stdin) ;
     }
    16 : printf("value of nl : %d",nl); 
    17 : printf("value of nw : %d",nw); 
    18 : printf("value of nc : %d",nc); 
}
```

Figure 1: Code for WordCount program

2.1 Control Flow Graphs

A Control Flow Graphs \(( CFG )\) is a graph representing the flow of control of a program and which can be defined as follows:

\[
CFG = (V, E)
\]

In this structure, each node in \( V \) is associated with a statement in the program and each edge in \( E \) connecting nodes of \( V \) represents a possible flow of control between statements.

The \( CFG \) for Word Count program is given in figure 2.

2.2 Dominance and post-dominance relations

Let \( G = (V, E) \) be a Control Flow Graph with unique entry node \( v_{hi} \in V \) and unique exit
node \( v_{out} \in V \) such that
\[
v_{in} = \{ v \in V \mid \exists w, v \in E \}
\]
and
\[
v_{out} = \{ v \in V \mid \exists w, w \in E \}
\]
We can say that:
\[
v_1 \ dom \ v_2 \Leftrightarrow \forall p = v_{in}, \ldots, v_2 >, v_1 \in p \quad (1)
\]
and
\[
v_1 \ pdom \ v_2 \Leftrightarrow \forall p = v_2, \ldots, v_{out} >, v_1 \in p \quad (2)
\]
for arbitrary nodes \( v_1 \in V \) and \( v_2 \in V \).

In other words, node \( v_1 \in V \) dominates node \( v_2 \in V \) if and only if every path in \( G \) from \( v_{in} \) to \( v_2 \) goes through \( v_1 \), and node \( v_1 \in V \) post-dominates node \( v_2 \in V \) if and only if every path in \( G \) from \( v_2 \) to \( v_{out} \) goes through \( v_1 \).

It has to be noted that some authors [7] restrict the post-dominance function not to be reflexive, i.e., \( v \ dom \ v \), but \( \neg (v \ pdom \ v) \). For the purposes of this paper, the distinction is not necessary, so we'll carry on the reflexive property of the relation \( pdom \).

Dominance and post-dominance relations [9, 10, 12, 13] can be refined to express immediate dominance \( idom \) and immediate post-dominance \( ipdom \). A node \( v_1 \) immediately dominates node \( v_2 \) if \( v_1 \) dominates \( v_2 \) and there is no other dominator of \( v_2 \) between \( v_1 \) and \( v_2 \) in the \( CFG \). More formally,
\[
v_1 \ idom \ v_2 \Leftrightarrow \alpha (v_1 \ dom \ v_2) \wedge (\forall w \in V, (w \ dom \ v_2) \wedge (w \neq v_2) \Rightarrow (w \ dom \ v_1)) \quad (3)
\]

Immediate post-dominance is a symmetrical relation to the one just expressed. A node \( v_1 \) immediately post-dominates node \( v_2 \) if \( v_1 \) post-dominates \( v_2 \) and there is no other post-dominator of \( v_2 \) between \( v_1 \) and \( v_2 \) in the \( CFG \). More formally,
\[
v_1 \ ipdom \ v_2 \Leftrightarrow \alpha (v_1 \ pdom \ v_2) \wedge (\forall w \in V, (w \ pdom \ v_2) \wedge (w \neq v_2) \Rightarrow (w \ pdom \ v_1)) \quad (4)
\]

\[\text{2.3 Dominator and post-dominator trees}\]

Given a Control Flow Graph \( G = (V, E) \), immediate dominance relations between nodes of \( V \) can be expressed in a tree structure, called a dominator tree \( (dom \ free) \). In this tree, the nodes are arranged in such a way that children nodes are immediately dominated by their direct parent node. More formally, let \( dom \ free = (V, E_D) \) be the dominator tree associated with \( G \), the set \( E_D \) of edges can be defined as:
\[
E_D = \{(v_1, v_2) \mid v_1 \ idom \ v_2 \} \quad (5)
\]
Post-dominance relations can also be expressed in a tree structure similar to \textit{dom}tree. We call $\text{pdom\_tree} = (V, E_{PD})$ the post-dominator tree computed for a given graph $G = (V, E)$. The set of edges $E_{PD}$ is defined as:

$$E_{PD} = \{(w_1, w_2) \mid w_1 \in \text{ipdom}w_2\} \quad (6)$$

### 2.4 Unconstrained Nodes

The set of unconstrained nodes $(UN)$ in a Control Flow Graph $G = (V, E)$ is defined as the set of nodes that dominate no other nodes and that post-dominate no other nodes, which corresponds to the following equation:

$$v_i \in UN \Leftrightarrow \begin{cases} \exists v_j \in V \mid (\forall p = v_i, ..., v_j, v_i \in p) \wedge \\
(\exists v_k \in V \mid (\forall q = v_k, ..., v_out, v_k \in q)) \end{cases} \quad (7)$$

where $v_i \in V$ is an node in $CFG$, $v_{in}$ and $v_{out}$ are the starting and ending nodes in $CFG$.

Note that this definition of unconstrained nodes is equivalent to the similar concept based on super blocks defined in [1].

The set $UN$ therefore represents a reduced set of nodes which have to be covered in order to achieve total node coverage of the program to be analyzed.

Given the previous definitions, the set $UN$ can be computed as the intersection of the leaves of the dominator tree of $CFG$ with the leaves of its post-dominator tree. See [1, 5, 6] for further details.

### 3 Data Flow Testing Coverage

Let $CFG = (V, E)$ be a control flow graph for a given program.

A definition $d$ is an identifier from the set $DEFJD$ that uniquely identifies a certain variable definition at a specific point in a program. Similarly a use $u$ from the set $REFJD$ uniquely identifies variable uses in a program.

Function

$$\text{var} : DEFJD \cup REFJD \rightarrow VARJD$$

takes a definition or use identifier and returns the variable which is respectively defined or referred.

Function

$$\text{node} : DEFJD \cup REFJD \rightarrow V$$

takes a definition or use identifier and returns the $CFG$ node where the definition or use occurs.

In presence of programming languages which allow pointers, the following functions can be defined:

$$PDEFS : V \rightarrow \mathcal{P}(DEFJD)$$
$$DDEFS : V \rightarrow \mathcal{P}(DEFJD)$$
$$PREFS : V \rightarrow \mathcal{P}(REFJD)$$
$$DREFS : V \rightarrow \mathcal{P}(REFJD) \quad (8)$$

Function $PDEFS$ returns the set of possible definitions performed at a certain node in the $CFG$, while function $DDEFS$ returns the set of definite definitions.

Similarly, function $PREFS$ returns the set of possible references performed at a given node, while function $DREFS$ returns the set of definite references.

Possible and definite definitions derive from pointer analysis and are related to definitions which define fixed locations and definitions which define variable locations. The same classification is made for possible and definite references. Flow and context insensitive possible definitions and references sets are computed for function calls even when a function pointer has to be dereferenced. These possible summaries are used for the intra-procedural analysis. See [15] for details about the pointer analysis equations used in conjunction with the approach presented in this paper.

A pair $<d, u>, d \in DEFJD, u \in REFJD$ is a generic definition-use pair if and only the following condition holds:
\[
(\exists p = \langle \text{node}(d), \ldots, \text{node}(u) \rangle \mid \\
((\forall v \in p, \text{node}(v) \neq \text{node}(d)) \Rightarrow \\
\text{var}(d) \notin \text{DDEF}(v)))))
\]

(9)

In other words a pair \(<d, u>\) is a def-use pair if and only if there exists a path in the Control Flow Graph from the node executing definition \(d\) to the node executing the reference \(u\). Furthermore, the variable defined by \(d\) must not be definitely redefined in the same path and it must coincide with the variable referenced by \(u\).

Table 1 presents the DDEFS and DREFS sets for the program “Word Count” whose source code is illustrated in Figure 1. Definitions and uses are presented in Table 1 in the following format:

\[
\begin{align*}
\text{DDEFS} : (d_{\text{node}}, d_{\text{id}}, <\text{proc}_{\text{id}}, \text{var}_{\text{id}} >) \\
\text{DREFS} : (u_{\text{node}}, u_{\text{id}}, <\text{proc}_{\text{id}}, \text{var}_{\text{id}} >)
\end{align*}
\]

(10)

where \(d_{\text{node}}\) is the definition node identifier, \(d_{\text{id}}\) is the definition identifier, \(u_{\text{node}}\) is the use node identifier, \(u_{\text{id}}\) is the use identifier, \(\text{proc}_{\text{id}}\) is the procedure identifier of the variable declaration (or \textit{GLOBAL} for global variables), and \(\text{var}_{\text{id}}\) is the variable identifier.

Some interesting nodes are nodes 9, 10, 11, and 14. Nodes 9, 11, and 14 constitute a set of unrestricted nodes, so that coverage of such a set covers the other nodes of the \textit{CFG}, if such a coverage is feasible.

It has to be noted that nodes 13 and 14 are equivalent from the point of view of determine the set \textit{UN} of unconstrained nodes, since they belong the the same basic block. Nodes 9, 11, and 13 would therefore constitute another set of unrestricted nodes. Refer to [1, 5] for further details.

Node 10 is interesting because three references are performed at such a node, namely references 4, 5, and 6, and the three of them involve the same variable \(c\) in function \textit{main}. Node 10 represents the boolean expression of a test which refers to multiple conditions on the same variable. Although multiple definitions are allowed in some C-language instructions, no nodes in “Word Count” presents them.

Nodes 9 and 14 present both a definition and a reference to the same variable, respectively \(<\text{main}, \text{nl}>\) at node 9 and \(<\text{main}, \text{nw}>\) at node 14. Node 11 defines variable \(<\text{main}, \text{inword}>\) and does not refer any variable.

<table>
<thead>
<tr>
<th>Word Count DDEFS and DREFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDEFS</td>
</tr>
<tr>
<td>[1, 0, &lt;main, inword &gt;]</td>
</tr>
<tr>
<td>[2, 1, &lt;main, nl &gt;]</td>
</tr>
<tr>
<td>[2, 3, &lt;main, nw &gt;]</td>
</tr>
<tr>
<td>[4, 4, &lt;main, c &gt;]</td>
</tr>
<tr>
<td>[10, 4, &lt;main, c &gt;]</td>
</tr>
<tr>
<td>[11, 7, &lt;main, inword &gt;]</td>
</tr>
<tr>
<td>[13, 8, &lt;main, inword &gt;]</td>
</tr>
<tr>
<td>[14, 9, &lt;main, nw &gt;]</td>
</tr>
<tr>
<td>[15, 10, &lt;main, c &gt;]</td>
</tr>
<tr>
<td>[16, 9, &lt;main, nl &gt;]</td>
</tr>
<tr>
<td>[17, 10, &lt;main, nw &gt;]</td>
</tr>
<tr>
<td>[18, 11, &lt;main, nc &gt;]</td>
</tr>
</tbody>
</table>

Table 1: DDEFS and DREFS sets for example program Word Count

### 3.1 Data Flow Coverage Metrics

Let \(p = \langle v_{i}^{1}, v_{j}^{2}, \ldots, v_{k}^{S} \rangle \) be a path in the \textit{CFG}, where \(v_{i}^{s}\) means node \(v_{i} \in V\) at position \(s\) in the path. We can define the functions \textit{POSS PATH COV} and \textit{DEF PATH COV} as follows:

\[
\text{POSS PATH COV} : P \times \text{DEF ID} \times \text{REF ID} \rightarrow \text{BOOLEAN} \\
\text{DEF PATH COV} : P \times \text{DEF ID} \times \text{REF ID} \rightarrow \text{BOOLEAN}
\]

(11)

where \(P\) is the set of all paths in program.

Function \(\text{POSS PATH L COV}(p, d, u)\) returns \text{TRUE} if \(v_{i}^{s} = \text{node}(d)\) precedes \(v_{j}^{s} = \text{node}(u)\) in \(p\) (i.e., if \(r < s\)), and if \(d, u >\) is a generic def-use in \(p\). In other words, \(p\) cov-
ers a def-use $<d, u>$ if and only if there are two nodes in the path for which the definition node precedes the use node and there is no definite redefinition of the defined variable between such two nodes in $p$.

Given a $CFG$, let the start node $v_0$ be represented as $v_0^1$, the exit node $v_{exit}$ be represented as $v_f^k$ where $s, f \in N$; let

$$p = <v_0^1, v_1^2, \ldots, v_i^{a-1}, v_f^k>$$

be a path, $d$ be a definition, and $u$ be a use. Function $POSS\_PATH\_COV(p, d, u)$ returns TRUE if the following condition holds:

$$(\exists v_d^b \in p) [(b < a) \land (v_d^b = \text{node}(d))] \land
(\exists v_u^c \in p) [(c \leq a) \land (v_u^c = \text{node}(u)) \land
(v = v_d^{b+1} \land \text{var}(d) = \text{var}(u))]) \land
(\forall v_m^e \in p, (b < e < c) \Rightarrow (v_m^e \neq \text{node}(d)) \land
\text{var}(d) \notin \text{DDEF}(v_m^e))]$$

Function $DEF\_PATH\_COV(p, d, u)$ returns TRUE if the following condition holds:

$$POSS\_PATH\_COV(p, d, u) \land
(\text{var}(d) \in \text{DDEF}(\text{node}(d))) \land
(\text{var}(u) \in \text{DDEF}(\text{node}(u)))$$

(13)

Equations 12 and 13 indicate that while possibly covered def-uses do not have any constraints, definitely covered def-uses must be formed only by definite definitions which match definite references. All other combination of definitions and uses may only give rise to possibly covered def-uses.

The concept of def-use coverage can be extended to nodes in the $CFG$. Semi-coverage functions $POSS\_SCOV(v)$ and $DEF\_SCOV(v)$ can be defined to capture the concept of defining the set of def-uses which can be respectively possibly or definitely covered by a path from the start node $v_0^1$ to a given node $v$ and from $v$ to the exit node $v_f^k$.

Semi-coverage functions are defined as follows:

$$POSS\_SCOV : V \rightarrow \mathcal{P}(DEF\_JD \times REF\_JD)$$

$$DEF\_SCOV : V \rightarrow \mathcal{P}(DEF\_JD \times REF\_JD)$$

(14)

$$POSS\_SCOV(v) = \{<d, u>| (\exists p =<v_0^1, v_2^2, \ldots, v_f^{a-1}, v_f^k> | (v = v_d^{b+1} \land \text{POSS}\_PATH\_COV(p, d, u)) \land
(v = v_1^1 \land \text{POSS}\_PATH\_COV(p, d, u))\}$$

(15)

$$DEF\_SCOV(v) = \{<d, u>| (\forall p =<v_0^1, v_2^2, \ldots, v_f^{a-1}, v_f^k> | (v = v_d^{b+1} \land \text{DEF}\_PATH\_COV(p, d, u)) \land
(v = v_1^1 \land \text{DEF}\_PATH\_COV(p, d, u))\}$$

(16)

Semi-coverage functions do not represent the node coverage for a given node $V$ since the def-uses $<d, u>$ where $(\text{node}(d) \prec v)$ and $(v \prec \text{node}(u))$, i.e. where the definition node strictly precedes $v$ in the $CFG$ and $V$ strictly precedes the use node, are not taken into account by equations 15 and 16.

Possible and definite node semi-coverage functions can be extended to entire routines by computing $POSS\_SCOV\_R$ and $DEF\_SCOV\_R$ functions as the union of node possible and definite coverage over all nodes in the $CFG$ of a routine, as follows:

$$POSS\_SCOV\_R = \bigcup v_i \in V \text{POSS}\_SCOV(v_i)$$

(17)

and definite def-use coverage is defined as:

$$DEF\_SCOV\_R = \bigcup v_i \in V \text{DEF}\_SCOV(v_i)$$

(18)
The following property holds:

\[
\{(v_1 \text{ dom } v_2) \lor (v_1 \text{ pdom } v_2)\} \Rightarrow \((\text{POSS}\_SCOV(v_1) \subseteq \text{POSS}\_SCOV(v_2)) \land (\text{DEF}\_SCOV(v_1) \subseteq \text{DEF}\_SCOV(v_2))\)
\]

Then possible and definite coverage can be also obtained as:

\[
\text{POSS}\_SCOVERAGE = \bigcup_{v_i \in UN} \text{POSS}\_SCOV(v_i)
\]

(19)

\[
\text{DEF}\_SCOVERAGE = \bigcup_{v_i \in UN} \text{DEF}\_SCOV(v_i)
\]

(20)

where \(UN\) is the set of unrestricted nodes.

Coverage functions \(\text{POSS}\_SCOVERAGE\) and \(\text{DEF}\_SCOVERAGE\) can be interpreted as static conservative bounds for the minimum data flow testing efforts based on node coverage. If we assume that it takes a distinct test case to cover a possible def-use and a distinct test case to cover a definite def-use, the two mentioned sets represent the conservative upper and lower bounds for the minimum number of data flow testing cases.

In reality, a test case may cover more than a single def-use, so, when statement coverage is achieved, usually more def-uses than simply the definite ones happen to be covered.

This paper presents a study on the possible and definite node and routine def-use semi-cover (\(\text{POSS}\_SCOV(v), \text{POSS}\_SCOVERAGE, \text{DEF}\_SCOV(v), \text{DEF}\_SCOVERAGE\)) obtainable by statement coverage on programs.

The comparison between static and dynamic figures is out of the scope of this paper and it is left for further research.

4 CANTO.

To run experiments using the proposed approach, a data set of C programming language capabilities (e.g., pointers, array, structures, function pointers, macros and so on) has been analyzed with CANTO (the Code and Architecture aNalysis T0ol), developed at IRST (Istituto per la Ricerca Scientifica e Tecnologica).

![Diagram](image)

Figure 3: Architecture of CANTO.

Figure 3 represents the architecture of CANTO. CANTO consists of a Front End module, which parses the source code, FLANT (the Flow ANalysis Tool), which computes several flow analyses on the code (e.g., data dependencies), ART (the Architectural Recovery Tool), which recognizes architectural patterns in the system under analysis, an Editor, which interacts with the user and highlights the results of the analyses directly on the source, and PROVIS (the PRogram VISualization tool), which displays the results as graphs, highlighting some nodes when required. A detailed description of the environment is presented in [3].

CANTO Front End produces the detailed information required to measure intra-procedural data flow testing coverage; it is based on Refine 1, and performs a very accurate transformation of the code into an intermediate representation, which accounts for complex pointer and array dereference chains, as well as for pointers and function pointers [15]. Code transformations aim to produce a semantically equivalent program based only on if, while and goto. The equivalent C program is then saved and a corresponding detailed Abstract Syntax Tree, Control Flow Graph intermediate representation are stored in a file together with definition, uses, and function calls solved with respect to static pointer analysis.

1 Refine and Refine/C are trademarks of Reasoning Systems Inc.
5 Experiments

The proposed approach has been implemented in C++ using Gnu compiler g++ version 2.8. Preliminary experiments have been carried out measuring the definite and possible semi-coverage of “Gnu find” tool whose size is about 16 KLOC on a Pentium Pro 180 MHz processor running Debian Linux, kernel 2.0.27, with 128 Mbytes RAM.

5.1 Example

In Appendix A, the sets POSS_COV(v) and DEF_COV(v) are presented for unconstrained nodes 9, 11, and 14 of the program “Word Count” listed in Fig. 1. Def-use pairs are represented as the following sequence:

\[(d_{node}, d_{id}, < proc_{id}, var_{id} >),\]
\[(u_{node}, u_{id}, < proc_{id}, var_{id} >)\]

(21)

where the symbols carry the same meaning explained for equation 10.

It has to be noted that nodes 13 and 14 are equivalent from the point of view of determine the set UN of unconstrained nodes, since they belong the the same basic block. Refer to [1, 5] for further details.

For the mentioned nodes, we can find in Appendix A the sets of possibly and definitely covered def-uses. Node 9, for example, may possibly cover def-use

\[(11, 7, < main, inword >),\]
\[(12, 7, < main, inword >)\]

(22)

because there exists a path

\[p = < 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,\]
\[15, 6, 7, 8, 10, 12,\]
\[15, 6, 16, 17, 18 >\]

in which variable \(< main, inword >\) is not redefined in any nodes in the following sub-path

\[q = < 15, 6, 7, 8, 10, 12 >\]

between nodes 11 and 12. Nevertheless, node 9 does not definitely cover the def-use presented in equation 22 because there exists a path

\[p = < 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,\]
\[15, 6, 16, 17, 18 >\]

in which such a def-use is not covered.

It can be observed that node 9 definitely covers def-use

\[(9, 6, < main, nl >), (16, 9, < main, nl >)\]

because all paths from node 9 to \(v_{out}\) contain a suffix path

\[q = < 9, ..., 15, 6, 16, 17, 18 >\]

in which \(< main, nl >\) is not redefined. Indeed, the mentioned definitely covered def-use corresponds to the propagation of the last update of the number of lines to the print statement.

With respect to def-uses involving variable \(< main, c >\), node 9 may possibly cover all of them, namely the following def-uses which correspond to the instruction that read a character (definitions) to the several references:

\[
< (5, 4, < main, c >), (6, 0, < main, c >) >
< (5, 4, < main, c >), (8, 2, < main, c >) >
< (5, 4, < main, c >), (10, 4, < main, c >) >
< (5, 4, < main, c >), (10, 5, < main, c >) >
< (5, 4, < main, c >), (10, 6, < main, c >) >
< (15, 10, < main, c >), (6, 0, < main, c >) >
< (15, 10, < main, c >), (8, 2, < main, c >) >
< (15, 10, < main, c >), (10, 4, < main, c >) >
< (15, 10, < main, c >), (10, 5, < main, c >) >
< (15, 10, < main, c >), (10, 6, < main, c >) >
\]

From the “definite” perspective, instead, only the following def-uses involving variable \(< main, c >\) are covered by node 9:

\[
< (5, 4, < main, c >), (6, 0, < main, c >) >
< (5, 4, < main, c >), (8, 2, < main, c >) >
< (15, 10, < main, c >), (6, 0, < main, c >) >
\]
Node 14, instead, also definitely covers the additional following def-uses involving variable <main,<c>:

\(<\{5,4,<main,<c>\},\{10,4,<main,<c>\}\rangle \>
\(<\{5,4,<main,<c>\},\{10,5,<main,<c>\}\rangle \>
\(<\{5,4,<main,<c>\},\{10,6,<main,<c>\}\rangle \>

It has to be noted that, although the definition node 5 and the use node 10 are the same for some of the def-uses involving variable <main,<c>>, they are considered distinct because distinct use identifiers are involved, as discussed in section 3.

In Table 2, cumulative results obtained from the analysis of word count are reported. In particular, in Table 2 the possible node def-use coverage (25 def-uses per node since they are all reachable) is reported.

The definite node def-use coverage distribution with respect to CFG nodes is also reported in terms of minimum possible coverage (18 def-uses per node), maximum possible coverage (25 def-uses per node), and average (about 22 def-use per node). Complete information about possible and definite semi-coverage for the program “Word Count” is reported in Appendix B.

<table>
<thead>
<tr>
<th>Word Count</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Possible (nodes)</td>
<td>Min</td>
</tr>
<tr>
<td></td>
<td>Average</td>
</tr>
<tr>
<td></td>
<td>Max</td>
</tr>
<tr>
<td>Definite (nodes)</td>
<td>Min</td>
</tr>
<tr>
<td></td>
<td>Average</td>
</tr>
<tr>
<td></td>
<td>Max</td>
</tr>
<tr>
<td>Possible (routine)</td>
<td>25</td>
</tr>
<tr>
<td>Definite (routine)</td>
<td>12</td>
</tr>
<tr>
<td>D/P ratio (%)</td>
<td>48</td>
</tr>
<tr>
<td>Execution time (%)</td>
<td>1.18</td>
</tr>
</tbody>
</table>

Table 2: Example results for program “Word Count”

Overall the whole routine, node coverage obtained by covering unrestricted nodes would possibly cover 25 def-uses and definitely cover 12 def-uses. To avoid any confusion, it has to be remarked that, with respect to the data presented in Appendices A and B, the cardinality of the union of DEF_COV sets is not the sum of the cardinalities.

The ratio between definitely and possibly overall covered def-uses compares the node coverage figures with respect to the “all uses” coverage ones. For “Word Count” this figure is 48%. The execution time has been slightly over one second.

5.2 Experimental Results

Preliminary experiments to assess the presented approach have been performed using “Gnu find” source code which has been pre-processed by CANTO to extract the Abstract Syntax Tree (AST), the Control Flow Graph, pointer information, and def-use chains as described in [3, 15].

All the routines composing the “find tool” have been extracted and their CFG has been processed according to the approach presented in this paper. In Table 3 cumulative results from all 115 routines identified in the “find tool” have been reported in terms of possible and definite intra-procedural coverage distribution, and ratio distribution between definitely and possibly covered def-uses for each routine. In such a table, we can observe that the average ratio is about 75%, which means that, on average, by covering the nodes of each routine, about four fifths of its def-uses are definitely covered. Furthermore, the maximum ratio is 100% which means that in some cases, covering the nodes of a routine implies covering all its def-uses.

A closer look to the routines lead the author to observe that indeed short routines carried a higher definitely to possibly covered ratio. It was not unusual to observe that for numerous short linear routines, covering all the statements implied also covering all semi-coverable intra-procedural def-uses. This ratio dropped rapidly, for longer and more complex routines.

Another experiment has been performed, to compute the distribution of definitely and possibly semi-covered sets over all the nodes of all the CFG for the different routines. The corresponding results are reported in Table 4.
Table 3: Cumulative results over all routines

<table>
<thead>
<tr>
<th>Routines</th>
<th>Min</th>
<th>Average</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Possible coverage</td>
<td>1</td>
<td>29.113</td>
<td>254</td>
</tr>
<tr>
<td>Definite coverage</td>
<td>1</td>
<td>12.443</td>
<td>108</td>
</tr>
<tr>
<td>D/P ratio (%)</td>
<td>0.395257</td>
<td>75.1903</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 4: Cumulative results over all nodes

<table>
<thead>
<tr>
<th>CFG Nodes</th>
<th>Min</th>
<th>Average</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Possible coverage</td>
<td>0</td>
<td>28.721</td>
<td>254</td>
</tr>
<tr>
<td>Definite coverage</td>
<td>1</td>
<td>5.52952</td>
<td>71</td>
</tr>
</tbody>
</table>

Although minimum and maximum values coincide - this need not necessarily be the case for all the systems - the average definite semi-coverage for each node is about 19% of average possible semi-coverage. This information, together with those reported in Table 3, give us a better portrait of the semi-coverage phenomenon.

Tables 5 and 6 respectively report data about the execution time of the experiments and the size of the experimental data. Reported times are CPU times for the approach presented in this paper, i.e. computing possible and definite semi-coverage for all routines and nodes. Preprocessing time performed by CANTO has been factored out. Execution times are reasonable and, although experiments on larger systems are needed to validate it, they indicate a performance which seems to be scalable.

Table 5: Performance

<table>
<thead>
<tr>
<th>Execution Time (sec)</th>
<th>Min</th>
<th>Average (routines)</th>
<th>Max</th>
<th>Average (nodes)</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.0899999</td>
<td>3.46269</td>
<td>11.245</td>
<td>0.12873</td>
<td>412.06</td>
</tr>
</tbody>
</table>

Table 6: Size of the experiments

<table>
<thead>
<tr>
<th>Size</th>
<th>LOC</th>
<th>Routines</th>
<th>Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16876</td>
<td>115</td>
<td>3201</td>
</tr>
</tbody>
</table>

5.3 Discussion

While the “all uses” possible semi-coverage metric appears to be less useful than the definite semi-coverage one because all reachable nodes in a CFG present the potential of reaching every use and therefore to possibly cover every def-use, the “all uses” definite semi-coverage metric could be useful in different ways.

First the measure itself gives an idea about how much “extra testing” has already been done for a given program while using a lower testing criterion. This measure is related to the topology and nature of a given program and it’s not a function of nor it depends on the test set. In this way programs can be characterized by their topological structure which links CFG nodes with definitions and uses through paths in the CFG. The proposed metric could be used to characterize the minimum “all uses testing yield” that the node coverage of a given program can give.

This measure can be used to rank programs for testing purposes. Programs, which show a higher node to “all uses” definite semi-coverage ratio, in some sort are better tested when node coverage testing criterion is achieved. Also, node coverage of these programs has a higher
chance of detecting defects which are related to errors in flow propagation and not only to statement execution alone.

On the other hand, programs with a lower value of the same metric may turn out weaker tested than other programs. The proposed semi-coverage metric can be used to identify such programs and perhaps additional testing using a testing criterion other than node coverage may be performed on these programs.

Additionally, given a particular test set for node coverage, the proposed metric could be used to set priorities among different test cases. A possible application of the presented metric would be to assign a priority to different test cases which is function of the metric value. An example could be of assigning a higher priority to test cases which increase both the number of covered nodes and the number of definitely semi-covered def-uses. In this way, both nodes and def-uses semi-coverage would increase rapidly while progressing in testing. Other priority schemes may be defined, too. A priority scheme can be used both by programmers to rearrange the order of execution of a particular test set and by tools for automatic generation of test data to guide the generation process based on the given priorities.

Obviously, extensive experimentation must be carried out in an industrial software development context to validate the practical usefulness of the proposed metric applications with respect to valuable software development objectives such as, for example, high software quality and reliability, low costs and high programmers’ productivity.

6 Conclusions and Further Research

An original approach to define and measure intra-procedural definite and possible def-use semi-coverage based on node coverage has been presented. Preliminary results obtained mainly on the freely available “Gnu find tool” have been presented and discussed.

Further research may involve validating the approach on larger systems in an industrial software development environment and on different programming languages. Semi-coverage functions may also be extended to take into account the coverage of def-use pairs for which the definition precedes a given node, which precedes the use.

The presented approach gives useful figures for unit testing, further development is necessary to extend the presented approach to system testing and inter-procedural def-use coverage analysis.

Also, since the presented approach completely relies on static analysis, it would be interesting to compare the static coverage findings with those dynamically obtained by performing statement coverage experiments.

7 Acknowledgments

This project has been partially funded by the National Science and Engineering Research Council of Canada (NSERC). The authors wish also to thank Sebastien Lapierre (Bell Canada) and Christian Meunier (Ecole Polytechnique of Montreal) for their contributions and the anonymous referees for their valuable comments.

8 About the Authors

Ettore M. Merlo received his Ph. D. in computer science from McGill University in 1989. He has been the lead researcher for software engineering at Computer Research Institute of Montreal until 1993 when he joined Ecole Polytechnique de Montreal where he is currently an associate professor and member of the CASI laboratory for design and analysis of software systems. His research interests are in software analysis, software reengineering, user interfaces, software maintenance and artificial intelligence. He has collaborated with industries and research centers in particular on software reengineering, clone detection, software quality assessment and architectural reverse engineering. He can be reached at the following address:

Department of Electrical and Computer Engineering
Ecole Polytechnique
P. O. Box 6079, Station Downtown,
Montreal, Quebec,
H3C 3A7, Canada
Phone: (514) 340-4711 ext. 5758
Giuliano Antoniol was born in Transacqua, Trento, Italy, on January 12, 1956. He received his doctoral degree in Electronic Engineering from the University of Padua in 1982. He is with Istituto per la Ricerca Scientifica e Tecnologica in Trento, Italy, where he is involved in several research and industrial projects, currently he leads the 1st Maintenance of Software System Project team. Results have been achieved in the architectural analysis of distributed software system and in C/C++ code analysis. His current research interests include software engineering, architectural recovery, reverse engineering, and distributed systems. He can be reached at the following address:

IRST-Istituto per la Ricerca Scientifica e Tecnologica,
Via alla cascata, Loc. Panté di Povo
38050 Povo (Trento), Italy
Phone: +39 (046) 314-524
Fax: +39 (0461) 314-591
E-mail: antoniol@ieee.org
URL: http://zeus.itc.it:4444/~antoniol/

REFERENCES


A Appendix

MODE: 9
POSS SCOV:

\[ \langle 11, 7, \langle \text{main, inword} \rangle \rangle, \langle 12, 7, \langle \text{main, inword} \rangle \rangle, \langle 9, 6, \langle \text{main, n1} \rangle \rangle, \langle 9, 3, \langle \text{main, n1} \rangle \rangle, \langle 7, 5, \langle \text{main, n0} \rangle \rangle, \langle 18, 11, \langle \text{main, n0} \rangle \rangle, \langle 5, 4, \langle \text{main, c} \rangle \rangle, \langle 6, 0, \langle \text{main, c} \rangle \rangle, \langle 15, 10, \langle \text{main, c} \rangle \rangle, \langle 4, 3, \langle \text{main, n0} \rangle \rangle, \langle 7, 1, \langle \text{main, c} \rangle \rangle \]

DEF SCOV CARDINALITY: 22

(11, 7, <main, inword>), (12, 7, <main, inword>), (9, 6, <main, n1>), (9, 3, <main, n1>), (7, 5, <main, n0>), (18, 11, <main, n0>), (5, 4, <main, c>), (6, 0, <main, c>), (15, 10, <main, c>), (4, 3, <main, n0>), (7, 1, <main, c>)

MODE: 11
POSS SCOV:

\[ \langle 11, 7, \langle \text{main, inword} \rangle \rangle, \langle 12, 7, \langle \text{main, inword} \rangle \rangle, \langle 9, 6, \langle \text{main, n1} \rangle \rangle, \langle 9, 3, \langle \text{main, n1} \rangle \rangle, \langle 7, 5, \langle \text{main, n0} \rangle \rangle, \langle 18, 11, \langle \text{main, n0} \rangle \rangle, \langle 5, 4, \langle \text{main, c} \rangle \rangle, \langle 6, 0, \langle \text{main, c} \rangle \rangle, \langle 15, 10, \langle \text{main, c} \rangle \rangle, \langle 8, 2, \langle \text{main, c} \rangle \rangle, \langle 15, 10, \langle \text{main, c} \rangle \rangle, \langle 10, 4, \langle \text{main, c} \rangle \rangle, \langle 15, 10, \langle \text{main, c} \rangle \rangle, \langle 10, 5, \langle \text{main, c} \rangle \rangle, \langle 15, 10, \langle \text{main, c} \rangle \rangle, \langle 10, 6, \langle \text{main, c} \rangle \rangle, \langle 2, 1, \langle \text{main, n1} \rangle \rangle, \langle 9, 3, \langle \text{main, n1} \rangle \rangle \]

DEF SCOV CARDINALITY: 6

MODE: 13
POSS SCOV:

\[ \langle 11, 7, \langle \text{main, inword} \rangle \rangle, \langle 12, 7, \langle \text{main, inword} \rangle \rangle, \langle 9, 6, \langle \text{main, n1} \rangle \rangle, \langle 9, 3, \langle \text{main, n1} \rangle \rangle, \langle 7, 5, \langle \text{main, n0} \rangle \rangle, \langle 18, 11, \langle \text{main, n0} \rangle \rangle, \langle 5, 4, \langle \text{main, c} \rangle \rangle, \langle 6, 0, \langle \text{main, c} \rangle \rangle, \langle 15, 10, \langle \text{main, c} \rangle \rangle, \langle 8, 2, \langle \text{main, c} \rangle \rangle, \langle 15, 10, \langle \text{main, c} \rangle \rangle, \langle 10, 4, \langle \text{main, c} \rangle \rangle, \langle 15, 10, \langle \text{main, c} \rangle \rangle, \langle 10, 5, \langle \text{main, c} \rangle \rangle, \langle 15, 10, \langle \text{main, c} \rangle \rangle, \langle 10, 6, \langle \text{main, c} \rangle \rangle, \langle 2, 1, \langle \text{main, n1} \rangle \rangle, \langle 9, 3, \langle \text{main, n1} \rangle \rangle \]

(14, 9, <main, nw>), (14, 8, <main, nw>)
POSS SCOV CARDINALITY: 22

DEF SCOV:

<11, 7, <main, inword>>, (12, 7, <main, inword>>
<9, 6, <main, n1>>, (9, 3, <main, n1>>
<9, 6, <main, n1>>, (16, 9, <main, n1>>
<7, 5, <main, nc>>, (7, 1, <main, nc>>
<5, 4, <main, c>>, (6, 2, <main, c>>
<5, 4, <main, c>>, (10, 4, <main, c>>
<5, 4, <main, c>>, (10, 5, <main, c>>
<4, 3, <main, nc>>, (7, 1, <main, nc>>
<15, 10, <main, c>>, (6, 0, <main, c>>

DEF SCOV CARDINALITY: 7

MODE: 14

POSS SCOV:

<11, 7, <main, inword>>, (12, 7, <main, inword>>
<9, 6, <main, n1>>, (9, 3, <main, n1>>
<9, 6, <main, n1>>, (16, 9, <main, n1>>
<7, 5, <main, nc>>, (7, 1, <main, nc>>
<5, 4, <main, c>>, (6, 0, <main, c>>
<5, 4, <main, c>>, (10, 4, <main, c>>
<5, 4, <main, c>>, (10, 5, <main, c>>
<4, 3, <main, nc>>, (7, 1, <main, nc>>
<3, 2, <main, nw>>, (14, 8, <main, nw>>
<15, 10, <main, c>>, (6, 0, <main, c>>
<15, 10, <main, c>>, (10, 4, <main, c>>
<15, 10, <main, c>>, (10, 5, <main, c>>
<12, 1, <main, n1>>, (9, 3, <main, n1>>
<14, 9, <main, nw>>, (14, 8, <main, nw>>
<14, 9, <main, nw>>, (17, 10, <main, nw>>

POSS SCOV CARDINALITY: 22

DEF SCOV:

<5, 4, <main, c>>, (6, 0, <main, c>>
<5, 4, <main, c>>, (8, 2, <main, c>>
<5, 4, <main, c>>, (10, 4, <main, c>>
<5, 4, <main, c>>, (10, 5, <main, c>>
<5, 4, <main, c>>, (10, 6, <main, c>>
<4, 3, <main, nc>>, (7, 1, <main, nc>>
<3, 2, <main, nw>>, (14, 8, <main, nw>>
<15, 10, <main, c>>, (6, 0, <main, c>>
<14, 9, <main, nw>>, (17, 10, <main, nw>>

DEF SCOV CARDINALITY: 9

B Appendix

POSSIBLE SCOVERAGE --->

(11, 7, <main, inword>>, (12, 7, <main, inword>>
(9, 6, <main, n1>>, (9, 3, <main, n1>>
(9, 6, <main, n1>>, (16, 9, <main, n1>>
(7, 5, <main, nc>>, (7, 1, <main, nc>>
(5, 4, <main, c>>, (6, 0, <main, c>>
(5, 4, <main, c>>, (6, 2, <main, c>>
(5, 4, <main, c>>, (10, 4, <main, c>>
(5, 4, <main, c>>, (10, 5, <main, c>>
(5, 4, <main, c>>, (10, 6, <main, c>>
(4, 3, <main, nc>>, (7, 1, <main, nc>>
(4, 3, <main, nc>>, (18, 11, <main, nc>>
(3, 2, <main, nw>>, (14, 8, <main, nw>>
(3, 2, <main, nw>>, (17, 10, <main, nw>>
(15, 10, <main, c>>, (6, 0, <main, c>>
(15, 10, <main, c>>, (6, 2, <main, c>>
(15, 10, <main, c>>, (10, 4, <main, c>>
(15, 10, <main, c>>, (10, 5, <main, c>>
(14, 9, <main, nw>>, (14, 8, <main, nw>>
(14, 9, <main, nw>>, (17, 10, <main, nw>>
(1, 0, <main, inword>>, (12, 7, <main, inword>>
(13, 8, <main, inword>>, (12, 7, <main, inword>>

DU_CARD: 25

<--- POSSIBLE SCOVERAGE

DEFINITE SCOVERAGE --->

(9, 6, <main, n1>>, (16, 9, <main, n1>>
(7, 5, <main, nc>>, (18, 11, <main, nc>>
(5, 4, <main, c>>, (6, 0, <main, c>>
(5, 4, <main, c>>, (6, 2, <main, c>>
(5, 4, <main, c>>, (10, 4, <main, c>>
(5, 4, <main, c>>, (10, 5, <main, c>>
(5, 4, <main, c>>, (10, 6, <main, c>>
(4, 3, <main, nc>>, (7, 1, <main, nc>>
(3, 2, <main, nw>>, (14, 8, <main, nw>>
(15, 10, <main, c>>, (6, 0, <main, c>>
(2, 1, <main, n1>>, (9, 3, <main, n1>>
(14, 9, <main, nw>>, (17, 10, <main, nw>>

DU_CARD: 12

<--- DEFINITE SCOVERAGE