UNIVERSITY OF SÃO PAULO

School of Arts, Sciences and Humanities

Technical Report PPgSI-001/2013

Proof of Correctness of the Bitwise Algorithm for Intra-procedural Data-flow Testing Coverage

Marcos Lordello Chaim
Roberto Paulo Andrioli de Araujo

January - 2013

Technical Report Series

PPgSI-EACH-USP. Arlindo Béttio St. 1000 - Ermelino Matarazzo - 03828-000.
São Paulo, SP. Brazil.
TEL: 55 (11) 3091-8197 http://www.each.usp.br/ppgsi

Proof of Correctness of the Bitwise Algorithm for Intra-procedural Data-flow Testing Coverage

Marcos Lordello Chaim[1], **Roberto Paulo Andrioli de Araujo**[1]

[1]Software Analysis and Experimentation Group (SAEG)
School of Arts, Sciences and Humanities
University of São Paulo
São Paulo – SP, Brazil

{chaim,roberto.araujo}@usp.br

Abstract. *Intra-procedural data flow testing coverage data can be very expensive to collect, especially for long running programs tracking the assignment of a value to a variable and its subsequent use,i.e., a definition-use association (dua). Recently, a novel algorithm, called Bitwise Algorithm (BA), which utilizes efficient bitwise operations and inexpensive data structures to track intra-procedural duas, has been proposed. BA's RAM memory requirements are restricted to three bit vectors the size of the number of duas. In terms of time complexity, its performance is at least as good as the most efficient previous algorithms and can be up to 100% more efficient. We present BA's proof of correctness showing that it correctly determines the duas covered during the program execution.*

1. Introduction

Data-flow (DF) testing provides comprehensive structural testing. It involves the development of tests which exercise every value assigned to a variable and its subsequent references (uses) occurring either in a computation or in a predicate. These entities are called *definition-use associations* (duas) [Rapps and Weyuker 1985]. Nevertheless, more than thirty years after its introduction, DF testing is hardly used in industrial settings. This situation occurs because DF testing tends to demand that many entities be verified. As a result, much effort is required to create test sets and to track duas at run-time.

To reduce the overhead due to tracking duas at run-time, we have proposed the bitwise algorithm (BA) for intra-procedural data-flow testing coverage [Chaim and de Araujo 2013]. BA utilizes efficient bitwise operations and inexpensive data structures to track intra-procedural duas. RAM memory requirements are restricted to three bit vectors the size of the number of duas. Conservative simulations indicate that the new algorithm imposes less execution slowdown.

More details on the description of the algorithm and the analyses and simulations carried out can be found in [Chaim and de Araujo 2013]. In what follows, we present the BA's proof of correctness showing that it determines correctly the duas coverage during the execution of a test case. The rest of this document is organized as follows. In Section 2, we briefly present the concepts utilized to model a program and the definition of DF testing entities. The bitwise algoritm (BA) is described in Section 3. In Section 4, the proof is presented. We draw our conclusions in Section 5.

2. Background

Let P be a program mapped into a flow graph $G(N, E, s, e)$ where N is the set of blocks of statements (nodes) such that once the first statement is executed all statements are executed in sequence, s is the start node, e is the exit node, and E is the set of edges (n',n), such that $n' \neq n$, which represents a possible transfer of control between node n' and node n. A *path* is a sequence of nodes (n_i, ..., n_k, n_{k+1}, ..., n_j), where $i \leq k < j$, such that $(n_k, n_{k+1}) \in E$.

Data-flow testing requires that selected test cases exercise paths in a program between every point a value is assigned to a variable and its subsequent references. When a variable receives a new value, it is said that a *definition* has occurred; a *use* of a variable happens when its value is referred to. A distinction is made between a variable referred to compute a value and to compute a predicate. When referred to in a predicate computation, it is called a *p-use* and is associated with edges; otherwise it is called a *c-use* when associated with nodes. A *definition-clear* path with respect to (wrt) a variable X is a path where X is not redefined in any node in the path, except possibly in the first and last ones.

Data-flow testing criteria in general require that *definition-use associations* (duas) be covered. The triple D = (d, u, X), called c-use dua, represents a data-flow testing requirement involving a definition in node d and a c-use in node u of variable X such that there is a definition-clear path wrt X from d to u. Likewise, the triple D = (d, (u', u), X), called p-use dua, represents the association between a definition and a p-use of a variable X. In this case, a definition-clear path (d,. . .,u',u) wrt X should exist.

The all uses data-flow testing criterion [Rapps and Weyuker 1985] requires the set of paths executed by the test cases of a test set T to include a definition-clear path for each dua (d, u, X) or (d, (u', u), X) of a program P. A test set with such a property is said to be *adequate* to the all uses criterion for program P since all required duas were *covered*.

3. Bitwise dua coverage algorithm

The bitwise algorithm (BA) for dua coverage is based on sets associated with each node of the flow graph. Below we formally define the sets of born (**Born**(n)), disabled (**Disabled**(n)), potentially covered (**PotCovered**(n)) and sleepy (**Sleepy**(n)) duas for a node $n \in N$ of a flow graph $G(N, E, s, e)$:

Born(n) : set of duas (d,u, X) or (d,(u',u), X) such that $d = n$.

Disabled(n) : set of duas (d,u, X) or (d,(u',u), X) such that X is defined in n and $d \neq n$.

PotCovered(n) : set of duas (d,u, X) or (d,(u',u), X) so that $u = n$.

Sleepy(n) : set of duas (d,(u',u), X) such that $u' \neq n$.

To determine the covered duas, BA keeps track of three working sets—the alive duas (**Alive**), the current sleepy duas (**CurSleepy**) and the covered duas (**Covered**). How these extra sets are determined is described in Algorithm 1.

```
Input: nodes traversed during program execution; sets Disabled($n$), Sleepy($n$),
       PotCovered($n$), and Born($n$)
Output: Covered set
1  Alive = $\varnothing$;
2  CurSleepy = $\varnothing$;
3  Covered = $\varnothing$;
4  repeat
5      $n$ = node traversed in program execution;
6      Covered = Covered $\bigcup$ [[Alive - CurSleepy] $\bigcap$ PotCovered($n$)];
7      Alive = [Alive - Disabled($n$)] $\bigcup$ Born($n$);
8      CurSleepy = Sleepy($n$);
9  until program execution finishes ;
10 return Covered
```

Algorithm 1: Bitwise dua coverage algorithm.

4. Correctness proof of the bitwise algorithm

Theorem 1. *Algorithm 1 is correct, that is, the Covered set contains the duas exercised in path $s \dots l$ where s is the start node and l is the last node traversed in a program execution.*

Proof. We prove the correctness of Algorithm 1 by induction.

Basis. The base case consists of determining the values of **Covered**, **Alive**, and **CurSleepy** for a path with a single node traversed and equal to s, the start node. For such a path, **Covered** is empty because duas (d,s,X) or (d,(u',s),X) do not exist. **Alive** contains duas (s,u,X) and (s,(u',u),X) which are birthed in s and **CurSleepy** comprises all p-use duas, except the p-use duas (s,(s,u),X).

Basis proof. After line 6, **Covered** is empty because the result of **Alive** minus **CurSleepy** is intersected with **PotCovered(**s**)** whose value is empty because there is not a dua with a use in s. **Alive** is set up with duas (s,u,X) and (s,(u',u),X) after line 7 since its initial empty value is added to **Born(**s**)**, which contains duas (s,u,X) and (s,(u',u),X). **CurSleepy** will receive the **Sleepy(**s**)**, determined according to the definition of **Sleepy(**n**)** in Section 3, after line 8. Thus, the basis is proved.

Induction. Assuming that **Covered** contains the exercised duas and **Alive** the *alive* duas in path ($s \dots n_k$) and **CurSleepy** contains the value of **Sleepy(**n_k**)**, then, after node n_{k+1} is processed in Algorithm 1, **Covered** will contain the exercised duas and **Alive** the alive duas in path ($s \dots n_k, n_{k+1}$) and **CurSleepy** will contain the value of **Sleepy(**n_{k+1}**)** .

Induction proof. Firstly, we show that **Covered** is correct after n_{k+1} is processed regarding only c-use duas. Following the inductive step, **Covered** and **Alive** are correct in path ($s, \dots, n_k$). At line 6 of Algorithm 1, the new **Covered** will contain the previous **Covered** added with the intersection of **Alive** and **PotCovered(**n_{k+1}**)**, which contains c-use duas (d,n_{k+1},X), according to the definition in Section 3. **CurSleepy** does not influence the result of **Covered** regarding c-use duas since it contains just p-use duas. Hence, **Covered** will contain the covered c-use duas up to n_k plus c-use duas (d,n_{k+1},X)

that are present both in **Alive** and **PotCovered**(n_{k+1}); that is, those c-use duas that are covered when path (s, . . .,n_k, n_{k+1}) is traversed.

The rationale with respect to p-use duas is similar. At line 6, **CurSleepy** is subtracted from **Alive**. However, **CurSleepy** is equal to **Sleepy**(n_k) before n_{k+1} is processed due to the inductive step. This implies **CurSleepy** contains all duas (d, (u',u), X) such that $u' \neq n_k$, following the definition of **Sleepy**(n) in Section 3. In other words, only p-use duas (d,(n_k,u), X) *are not* in **CurSleepy** before n_{k+1} is processed. Thus, (d,(n_k,u), X) are the only p-use duas left after the subtraction of **CurSleepy** from **Alive**. P-use duas (d,(n_k,u), X) are then intersected with **PotCovered**(n_{k+1}), which contains p-use duas (d,(u',n_{k+1}),X) (see definition in Section 3). As a result, **Covered** will contain the covered p-use duas up to n_k plus p-use duas (d,(n_k,n_{k+1}),X) that are present both in **Alive** and in **PotCovered**(n_{k+1}); that is, those p-use duas that are covered when path (s, . . .,n_k, n_{k+1}) is traversed.

To complete the induction proof, however, it needs to be shown that **Alive** and **CurSleepy** are correct after n_{k+1} is processed. Considering the **Alive** working set, its value is correct up to n_k before executing line 7 of the algorithm. At line 7, duas (d,u, X) or (d,(u',u), X), such that X is defined in n_{k+1} and $d \neq n_{k+1}$, are eliminated from **Alive** because they belong to **Disabled**(n_{k+1})—see definition of **Disabled**(n) in Section 3. This subtraction operation represents the redefinition of variable X at n_{k+1}. The subsequent union with **Born**(n_{k+1}) which contains duas (n_{k+1},u,X) and (n_{k+1},(u',u),X), adds to **Alive** those duas created from the definition of X at n_{k+1}. Thus, after line 7, **Alive** contains the duas *alive* in path (s, . . .,n_k, n_{k+1}). At line 8, **CurSleepy** receives the value of **Sleepy**(n_{k+1}). Hence, the induction is proved.

After $n_{k+1} = l$ is processed, **Covered** will contain the set of duas exercised in path ($s \dots l$), which proves Theorem 1. □

5. Conclusions

The proof of correctness of the bitwise algoritm (BA) shows that it correctly determines the definition use associations covered during the execution of a program. The algorithm is correct even when the program execution terminates without traversing the exit node e. For example, if there is an abort command in a node n such that $n \neq e$, the program will terminate its execution at n, before reaching the exit node. Nonetheless, the proof of correctness presented shows that BA correctly determines the covered duas.

References

Chaim, M. L. and de Araujo, R. P. A. (2013). An efficient bitwise algorithm for intra-procedural data-flow testing coverage. *Information Processing Letters*. To appear.

Rapps, S. and Weyuker, E. J. (1985). Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-11(4):367–375.