

On data flow testing strategy

T.Y. Chen & J.Q. Mao

*Department of Computer Science, University of
Melbourne, Parkville 3052, Australia*

Abstract

For the studies on the data flow oriented selection of test data [4], [5], [6], [8], the selection strategies are all based on the intuition that an incorrect definition could never be uncovered if its use is not tested. In this paper, we are proposing another data flow oriented selection strategy based on the intuition that errors tend to be clustered together and are more likely to be associated with the improper or bad programming practice. It has been found that our strategy is complementary to [4], [5], [6] and [8], and can be incorporated at nominal overheads with Laski and Korel's strategy [5].

INTRODUCTION

The technique of data flow analysis was originally used in compiler object code optimization [1]. Then, it has been used in program testing, to detect data flow anomalies [2], [7], and to select test data [4], [5].

Most test data selection strategies are control flow oriented, that is, based on the control structures of the program. It has been suggested in [4], [5], [6], [8] that, apart from control flow information, data flow information should be considered as well since it is also a potential source of structural information about a program. Their strategies are all motivated by the intuition that an incorrect definition could never be uncovered if its use is not exercised. In this paper, we are going to present another selection strategy based on the intuition that errors tend to be clustered together and are more likely to be associated with the improper or bad programming practice. It



has been found that this selection strategy is complementary to the previously mentioned data flow oriented ones. Furthermore, its implementation could be easily incorporated at nominal overheads with that of Laski and Korel's strategy [5].

LASKI AND KOREL'S STRATEGY

In this section, we briefly introduce Laski and Korel's strategy. For details, readers should consult [5]. Basically, we follow their notations. The basic idea behind their strategy is to check the liveness of every definition of a variable at the points of its possible use, and exercise all definition—use chains.

We use the following example to illustrate the notions and concepts of their strategy.

statement no.	program sample;	definitions
	var x, y, z : integer;	
	begin	
1	read(x);	x^1
2	y := 1;	y^2
3	z := y;	z^3
4	while x > 0 do	
	begin	
5	if x - y > 0	
6	then y := y + 1	y^6
7	else x := 0;	x^7
8	z := y / x;	z^8
	end;	
9	write(z);	
	end.	

Figure 1: A sample program

Definitions of variables mean assigning values to the variables; and uses of variables mean reference of variables. For example, in the program in Figure 1, there are definitions of variables x and y in statements 1 and 2 respectively, and uses of variables x and y in statement 5. v^k is used to denote the definition of variable v in statement k . For example, z^8 is a

definition of z in statement 8.

The definition v^k of variable v is said to be live at statement m , if there exists an execution path from k to m along which the value of v assigned in k is not redefined by any other statements. The tuple (v, k, m) is called a *definition-use chain*, if v^k is live at m . For example, y^2 , the definition of y in statement 2, is live at statement 5; however, it is not live along the path $(1 - 2 - 3 - 4 - 5 - 6 - 8 - 9)$ at statement 8 because there is another definition of y , y^6 , on the path that redefines the value assigned to y in statement 2. Note that y^6 is not live at statement 3 because there is no execution path from the definition to that use.

A *data environment* is the set of all live definitions of all variables used (referenced) in statement i (denoted by $DE(i)$). Followings are the data environments of the sample program in Figure 1.

$$\begin{aligned} DE(3) &= \{y^2\}, \\ DE(4) &= \{x^1, x^7\}, \\ DE(5) &= \{x^1, y^2, x^7, y^6\}, \\ DE(6) &= \{y^2, y^6\}, \\ DE(8) &= \{x^1, y^2, x^7, y^6\}, \\ DE(9) &= \{z^3, z^8\}. \end{aligned}$$

However, if more than one variable is referenced in a statement in the program, these variables should be used together instead of individually. This fact leads to the concept of data context. For a statement i that uses a set of variables $V(i) = \{v_1, v_2, \dots, v_n\}$, $ec(i) = (v_1^{i1}, v_2^{i2}, \dots, v_n^{in})$ is defined as an elementary context, if there exists a control path from the beginning of the program to i and $v_1^{i1}, v_2^{i2}, \dots, v_n^{in}$ are live at i along this path. For example, $ec(8) = (x^1, y^6)$ is an elementary context of statement 8. The *data context* of statement i , $DC(i)$, is defined as the set of all its elementary contexts. Followings are all the data contexts of the sample program in Figure 1.

$$\begin{aligned} DC(3) &= \{(y^2)\}, \\ DC(4) &= \{(x^1), (x^7)\}, \\ DC(5) &= \{(x^1, y^2), (x^1, y^6), (x^7, y^2), (x^7, y^6)\}, \\ DC(6) &= \{(y^2), (y^6)\}, \\ DC(8) &= \{(x^1, y^2), (x^1, y^6), (x^7, y^2), (x^7, y^6)\}, \\ DC(9) &= \{(z^3), (z^8)\}. \end{aligned}$$

The data environment criterion requires that the liveness of each definition in the data environment of every statement be tested at least once, while the data context criterion requires that each elementary data context of every statement be tested at least once.



ANOTHER SELECTION STRATEGY

Motivation

Laski and Korel's strategy [5] is based on the intuition that an incorrect definition could not be uncovered if its uses are never tested. It requires to select test data that would exercise certain paths from a point in a program where a variable is defined, to points at which that variable is referenced.

As it is practically infeasible to test all paths, intuitively, we should give preference to test those paths which are more likely to have errors. It is a well-known experience that errors tend to be clustered together and are more likely to be associated with improper or bad programming practice. As pointed out in [3], even if an anomaly is not an error, it is usually due to bad programming practice. Therefore, it is natural to select test data to exercise all data flow anomalies. Since the anomalies are categorized as $d-d$ (define-define), $d-u$ (define-undefine) and $u-r$ (undefine-reference) anomalies, our strategy is to exercise all their corresponding chains, namely $d-d$, $d-u$ and $u-r$ chains, respectively.

$d-d$ chains

statement no.	program sample1;
	var x, y, z : integer;
	begin
1	y := 0;
2	z := 1;
3	read(x);
4	if x > 0
5	then y := -z { correct one is: y := y - z }
6	else y := y + z;
7	write(y);
	end.

Figure 2: A sample program showing $d-d$ anomaly

No	statement	(un)definition	DE(i)	DC(i)	D(i)
0	undefine actions	x_0, y_0, z_0			
1	$y := 0$	y^1			x_0, y_0, z_0
2	$z := 1$	z^2			x_0, y^1, z_0
3	read(x)	x^3			x_0, y^1, z^2
4	if $x > 0$		x^3	x^3	x^3, y^1, z^2
5	then $y := -z$	y^5	z^2	z^2	y^1, z^2
6	else $y := y + z$	y^6	y^1, z^2	(y^1, z^2)	z^2
7	write(y)		y^5, y^6	y^5, y^6	y^5, y^6
8	undefine actions	x_8, y_8, z_8			

Table 1: $DE(i)$, $DC(i)$ and $D(i)$ for Figure 2's program

We illustrate our strategy by considering the simple program shown in Figure 2, in which statement 5 should be $y := y - z$ instead of $y := -z$. In Table 1, apart from $DE(i)$ and $DC(i)$, we construct a new set, called $D(i)$, where i stands for a statement number. Note that two pseudo-statements 0 and 8 are assumed as the first and last statements of the program respectively. The first statement imposes undefine actions on all variables of the program as all variables are assumed to have undefined values before any initialization; while the undefine actions of the last statement is added to detect unused definitions before exit of the program. In order to distinguish between define actions and undefine actions, v_i and v^i are used to denote an undefine and define action respectively on variable v at statement i . By $D(i)$, we mean that for statement i , if the definition v^k or undefinition v_k of variable v is not referenced or redefined respectively along some paths from k to i , then v^k or v_k is an element of $D(i)$, with the exception that if statement i is an assignment statement of variable v which is also referenced at the right hand of statement i , then the definition v^k will not be included in $D(i)$.

Since $DE(i)$ contains the live definitions of the variables that are not redefined before the execution of statement i and $D(i)$ contains the unreferenced definitions and live undefinitions of the variables at statement i , so the constructions of $DE(i)$ and $D(i)$ are basically the same with the exception of checking that an action is a define one in $DE(i)$, but an action is a reference or undefine one in $D(i)$. Therefore, the construction of $D(i)$ can easily be computed along with the construction algorithm of $DE(i)$ at nominal cost.

For example, $D(3) = \{x_0, y^1, z^2\}$ means that from statement 0 to statement 3, both definitions y^1 and z^2 and undefinition x_0 may have not been

304 Software Quality Management

referenced or redefined; in other words, there may be unreferenced definitions or live undefinitions at statement 3 before the execution of statement 3. Similarly, $D(6) = \{z^2\}$ indicates that definition z^2 may be live at statement 5; but since variable y is referenced at the right hand of statement 6 which means definition y^1 is referenced at statement 3, so y^1 is not included in $D(i)$. In addition to $D(i)$, we construct $def(v)$ which is a set of statement numbers in the program where variable v is defined. For example, $def(y) = \{1, 5, 6\}$ means that, in the program, y is defined at statement 1, 5 and 6. Now we are going to present the concept of $d-d$ chain. The tuple (v, k, m) is defined a $d-d$ chain if there exists a path from k to m such that v^k is not yet referenced in m and there is a definition of v in m . It is obvious from the definition of $d-d$ chain that it has a $d-d$ anomaly.

Following is the algorithm for $d-d$ chain for variable v :

$d-d$ chain Algorithm:

```
while  $def(v) \neq \text{empty}$ 
    select  $m \in def(v)$ ;
     $def(v) = def(v) \setminus \{m\}$ ;
    for all  $d \in D(m)$ 
        if  $d$  is of the form  $v^k$ 
            then  $(v, k, m)$  is reported as a  $d-d$  chain;
    end;
end
```

Using this algorithm, we can construct the $d-d$ chains in the program. For example, since $5 \in def(y) = \{1, 5, 6\}$ and $y^1 \in D(5) = \{y^1, z^2\}$, so we know that there is a $d-d$ anomaly on variable y along the path from statement 1 to statement 5. However, this error cannot be detected by either the data environment or data context criterion.

$d-u$ chains

In Figure 3, suppose statement 5 is mistyped as $z := 2$. A $d-u$ anomaly on variable y occurs along the path $(1 - 2 - 3 - 5 - 6)$.

```
statement no.    program sample2;
                  var x, y, z : integer;
                  begin
1                 y := 0;
2                 read(x);
3                 if x > 0
4                   then z := y + 1
5                   else z := 2; { correct one is: z := y + 2 }
6                 write(z);
                  end.
```

Figure 3: A sample program showing $d-u$ anomaly

No	statement	(un)definition	DE(i)	DC(i)	D(i)
0	undefine actions	x_0, y_0, z_0			
1	$y := 0$	y^1			x_0, y_0, z_0
2	read(x)	x^2			x_0, y^1, z_0
3	if $x > 0$		x^2	x^2	x^2, y^1, z_0
4	then $z := y + 1$	z^4	y^1	y^1	z_0
5	else $z := 2$	z^5			y^1, z_0
6	write(z)		z^4, z^5	z^4, z^5	y^1, z^4, z^5
7	undefine actions	x_7, y_7, z_7			y^1

Table 2: $DE(i)$, $DC(i)$ and $D(i)$ for Figure 3's program

$undef(v)$ is defined as the set of statement numbers in the program where variable v is undefined. For instance, $undef(y) = \{0, 7\}$ means that variable y is undefined at statements 0 and 7.

The tuple (v, k, m) is defined as a $d-u$ chain for variable v , if there exists a path from k to m such that v is undefined at m and v^k is not yet referenced in m . Obviously, a $d-u$ chain contains a $d-u$ anomaly.

306 Software Quality Management

The algorithm of $d-u$ chain for variable v is given in the following:

$d-u$ chain Algorithm:

```
while undef( $v$ )  $\neq$  empty
    select  $m \in \text{undef}(v)$ ;
     $\text{undef}(v) = \text{undef}(v) \setminus \{m\}$ ;
    for all  $d \in D(m)$ 
        if  $d$  is of the form  $v^k$ 
            then ( $v, k, m$ ) is reported as a  $d-u$  chain;
    end;
end
```

For example, since $7 \in \text{undef}(y) = \{0, 7\}$ and $y^1 \in D(7) = \{y^1\}$, we therefore know that definition y^1 has not been used in some execution path in the program and a $d-u$ anomaly on y has occurred. Then, we should select such test data to execute this chain. Again, this error cannot be detected by either the data environment or data context criterion.

$u-r$ chains

In Figure 4, a $u-r$ anomaly occurs along the path (1-2-3-5-6-7) where statement 5 was mistyped as $y := 2$ while the correct one should be $z := 2$.

statement no.	program sample3;
	var x, y, z : integer;
	begin
1	y := 0;
2	read(x);
3	if x > 0
4	then z := 1
5	else y := 2; { correct one is: z := 2 }
6	y := y + z;
7	write(y);
	end.

Figure 4: A sample program showing $u-r$ anomaly

No	statement	definition	DE(i)	DC(i)	D(i)
0	undefine actions	x_0, y_0, z_0			
1	$y := 0$	y^1			x_0, y_0, z_0
2	read(x)	x^2			x_0, y^1, z_0
3	if $x > 0$		x^2	x^2	x^2, y^1, z_0
4	then $z := 1$	z^4			y^1, z_0
5	else $y := 2$	y^5			y^1, z_0
6	$y := y + z$	y^6	y^1, y^5, z^0, z^4	(y^1, z^4)	z_0
7	write(y)		y^6	y^6	y^6
8	undefine actions	x_8, y_8, z_8			

Table 3: $DE(i)$, $DC(i)$ and $D(i)$ for Figure 4's program

Before defining a $u-r$ chain for variable v , we first define $ref(v)$ which is the set of statement numbers in the program where variable v is referenced. For example, $ref(z) = \{6\}$ means that variable z is referenced at statement 6. A pair (v, k, m) is defined as a $u-r$ chain for variable v , if there exists a path from k to m such that v is undefined in k and v_k is referenced in m . Obviously, a $u-r$ anomaly occurs in a $u-r$ chain.

The algorithm of $u-r$ chain for variable v is as follows:

$u-r$ chain Algorithm:

```
while  $ref(v) \neq \text{empty}$ 
  select  $m \in ref(v)$ ;
   $ref(v) = ref(v) \setminus \{m\}$ ;
  for all  $d \in D(m)$ 
    if  $d$  is of the form  $v_k$ 
      then  $(v, k, m)$  is reported as a  $u-r$  chain;
  end;
end
```

Since $6 \in ref(z)$ and $z_0 \in D(6) = \{z_0\}$, a $u-r$ anomaly has occurred on variable z along the path (1-2-3-5-6). Obviously, this anomaly is an error.

308 Software Quality Management

Data context criterion can only check the definition–use chain of y and z along the path (1–2–3–4–6), but it cannot check another definition–use chain of y and z along the path (1–2–3–5–6) because only definition y^5 reaches statement 6 and the definition of variable z is not available. Thus, this error cannot be detected. But, data environment criterion is able to pick up the path (1–2–3–5–6), and thus is able to detect this error.

A pair (v, k, m) is defined as a *ur-related d-r* chain, if (v, j, m) is a *u-r* chain and there exists a path from k to m such that (v, k, m) is a *d-r* chain. With the intuition that errors tend to be clustered together, since the *ur* anomaly is definitely an error, *ur-related d-r* chains are therefore more likely to contain errors than the other *d-r* chains. Thus, *ur-related d-r* chains should be given preference to be tested first among the *d-r* chains.

Following is the algorithm of *ur-related d-r* chains for variable v :

ur-related d-r chain Algorithm:

Apply *u-r* chain Algorithm

($URC(v)$ = set of all *u-r* chains for v)

Repeat

if $(v, j, m) \in URC(v)$

then begin

$URC(v) = URC(v) \setminus \{(v, j, m)\}$

for all $d \in D(m)$

if d is of the form v^k

then (v, k, m) is reported as a *ur-related d-r* chains

end;

end;

until $URC(v) = \text{empty}$

CONCLUSION

In this paper, we have demonstrated that our strategy can detect some errors that could not be detected by using Laski and Korel's data environment and data context criteria. However, our intuition is complementary to and not conflicting with Laski and Korel's intuition. Also, as explained above, our $D(i)$ can be constructed along with the construction of their $DE(i)$ at nominal costs. We believe that it is worthwhile to integrate these two strategies together.

ACKNOWLEDGEMENT

This research was partially supported by an ARC Small Grant 1993.

References

- [1] Allen, F. E. 'A Basis for Program Optimization,' pp. 385-390, *Proceedings of IFIP Conference*, North Holland Publishing Company, Amsterdam, 1971.
- [2] Chan, F. T. and Chen, T. Y. 'AIDA - A Dynamic Data Flow Anomaly Detection System for Pascal Programs,' *Software - Practice and Experience*, Vol. 17, pp. 227-239, 1987.
- [3] Chen, T. Y. and Poole, P. C. 'Dynamic Dataflow Analysis,' *Information and Software Technology*, Vol. 30, pp. 497-505, 1988.
- [4] Herman, P. M. 'A Data Flow Analysis Approach to Program Testing,' *Australian Computer Journal*, Vol. 8, pp. 92-96, 1976.
- [5] Laski, J. W. and Korel, B. 'A Data Flow Oriented Program Testing Strategy,' *IEEE Transactions on Software Engineering*, Vol. 9, pp. 347-354, 1983.
- [6] Ntafos, S. C. 'On Required Element Testing,' *IEEE Transactions on Software Engineering*, Vol. 10, pp. 795-803, 1984.
- [7] Osterweil, L. J. and Fosdick, L. D. 'DAVE - A Validation Error Detection and Documentation System for Fortran Programs,' *Software - Practice and Experience*, Vol. 6, pp. 473-586, 1976.
- [8] Rapps, S. and Weyuker, E. J. 'Selecting Test Data Using Data Flow Information,' *IEEE Transactions on Software Engineering*, Vol. 11, pp. 365-375, 1985.