A Data Flow Oriented Program Testing Strategy

JANUSZ W. LASKI AND BOGDAN KOREL

Abstract—**Some properties of a program data flow can be used to guide program testing. The presented approach aims to exercise use-definition chains that appear in the program. Two such data oriented testing strategies are proposed; the first involves checking liveness of every definition of a variable at the point(s) of its possible use; the second deals with liveness of vectors of variables treated as arguments to an instruction or program block. Reliability of these strategies is discussed with respect to a program containing an error.**

Index Terms—**Control flow, data context, data environment, data flow, data oriented testing, program testing, liveness, variable definition.**

I. Introduction

WHILE planning a program testing procedure the following questions arise: 1) which parts of the program to test, 2) how to determine the program input data to exercise those parts, and 3) how to interpret the observed intermediate or final results to assess (in)correctness of the program and eventually locate an error(s).

Possible solutions to the first problem are known as *testing strategies*. The second and third questions are referred to as the test data *selection* and *evaluation* problems, respectively. While both the test-data selection and test evaluation problems fall into the category of semantic analysis, the testing strategy is often chosen on purely structural grounds. In most cases, however, such a strategy is defined in terms of the control structure of the program to be tested. This applies to actual testing as well as to symbolic testing [1]-[4]. Data flow analysis is another potential source of structural information about a program, which seems to have been unexplored, at least as far as actual testing is concerned. Clearly, there exist methods for detecting possible anomalies, or inconsistencies in program data flow, either through static analysis of the text of a program [6], [7] or through its run-time instrumentation [5]; however, data flow itself has not been used to guide the testing procedure except perhaps, its use as an error-localization tool [8].

In this paper an attempt is undertaken to employ some properties of data flow as a criterion for test path selection. The essential notions introduced are the *data environment* and *data context*. These notions are defined at the statement and block level of a program. A testing strategy can be determined to exercise those control paths along which all chosen elements of the data environment or data context are activated. Two such strategies are proposed in Section V and brief discussion on their reliability follows in Section VI. The first is based on the data environment and the second on the data context. These strategies can be applied either at the statement level or at the block level. The considerations are illustrated by an example of a program with an error (Section II). This is followed by application of some typical control oriented testing strategies to that program and discussion of their reliability (Section III).

Definitions: In the following, the control graph $G = (N, A)$ of a program is a directed, connected graph having a unique entry node, $en \in N$ and unique exit node, $ex \in N$. A node $n \in N$ represents either a single instruction (instruction level) or a block (block level). A *block* is a single-entry single-exit sequence of instructions which are always executed together. Every instruction in a block, with perhaps the exception of the first instruction, has exactly one predecessor and, with the exception of the last instruction, exactly one successor.

An arc $a \in A$ is a pair (n,m) of modes from N which represents a possible transfer of control from n to m. A *path* in the graph is an ordered sequence $a_1, a_2, \cdots, a_k$ of arcs from A such that 1) the first node in a_1 is en, 2) the last node in a_k is ex, 3) for any two adjacent arcs $a_i = (n,m)$ and $a_{i+1} = (n^1, m^1)$, $m = n^1$. A path is *executable* (or feasible) if there exists input data which causes the path to be traversed during program execution. Otherwise, the path is unexecutable (or *infeasible*).

II. An Example

The program in Fig. 1 is a modified version of a sorting program from [2]. The program is supposed to carry out a simple sort of array $a[0:N]$ in descending order. The inner loop around instructions i_6 through i_{10} finds the largest element of the subarray $a[R1:N]$. That element is then copied into $R0$ while $R3$ points to its location in a. The outer loop is supposed to swap $a(R1)$ and $a(R3)$; the completion of the outer loop body assures that the subarray $a[0:R1-1]$ has been sorted and $a[R1-1]$ is greater than or equal to any element of $a[R1:N]$.

The Error and its Manifestation: There is a missing reinitialization of $R3$ to $R1$ at the beginning of the inner loop. Therefore, if at the beginning of some iteration of the inner loop, $a(R1)$ is the largest element in $a[R1:N]$, then $R3$ is not reset and retains its old value. The old value can be that from a previous execution of the inner loop during which $R3$ was reset or the initial value of $R3$ if $a[0:R1-1]$ was initially sorted and its elements were not less than those in $a[R1:N]$.

Manuscript received December 22, 1980; revised March 8, 1982.

J. W. Laski is with the School of Engineering and Computer Science, Oakland University, Rochester, MI 48063.

B. Korel is with the School of Engineering and Computer Science, Oakland University, Rochester, MI 48063, on leave from the Institute of Control Systems, Katowice, Poland.

0098-5589/83/0500-0347$01.00

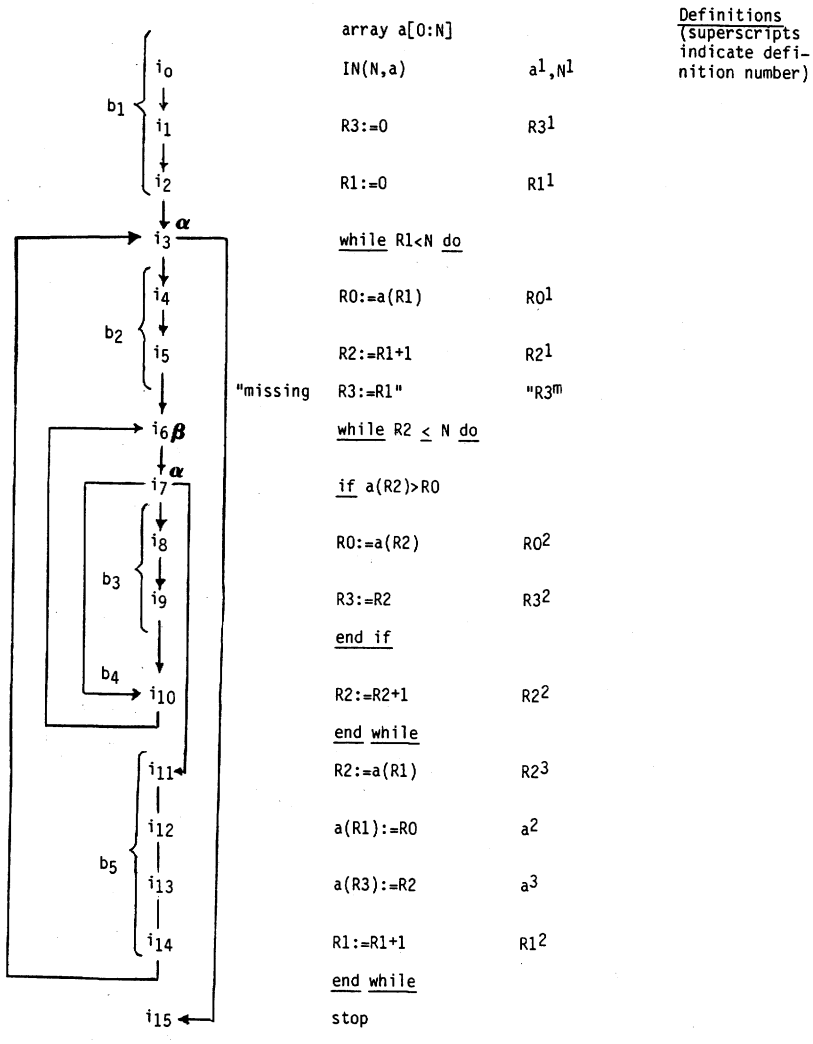


Fig. 1. A modified version of the sorting program, its control graph and definitions of variables associated with particular instructions; instruction i_1 not required if the missing instruction $R3 := R1$ is present.

Consequently, when $a(R1)$ and $a(R3)$ are being swapped by the sequence i_{11} through i_{13} the current value of $a(R1)$ overwrites $a(R3)$. Unless some conditions are met the element $a(R3)$ will be lost. To illustrate this phenomenon suppose that $N = 4$ and the initial value of a is (5, 7, 8, 1, 2). After the first iteration of the outer loop a becomes (8, 7, 5, 1, 2) but after the next complete execution of the inner loop there is $R3 = 2$, $R1 = 1$, and $R0 = a(R1) = 7$. Thus, the swapping sequence leads to $a = (8, 7, 7, 1, 2)$ and the initial entry $a(0) = 5$ will be lost. Note that under correct conditions (i.e., if $R3 := R1$ had existed) there would have been $R1 = R3 = 1$ and the swapping would have involved the same memory location. Finally, when the program terminates we have a sorted array (8, 7, 7, 2. 1) which, however, is not a permutation of the original one.[1]

In the original version of the program in [2], instruction $i_1 : R3 = 0$ is missing because it is not necessary if the program is correct, i.e., if reinitializing instruction $R3 := R1$ is not missing. However, a data flow anomaly analysis would perhaps detect this as an uninitialized variable being referenced. Therefore, to avoid this trivial case instruction i_1 has been inserted into the program. This is equivalent to the case when a compiler sets the initial values of certain types to zero; in such a case, however, a static code analysis would still issue a warning of a possible data flow anomaly, although no run-time anomaly would really occur.

Error Detection: The necessary although not sufficient condition for the error to be detected either upon the termination of the program or at some intermediate step of its execution is to provide input data which causes the control of the program to traverse a path which 1) at least twice iterates the outer loop and 2) before passing the segment i_{11} through i_{14} at some iteration of the outer loop, has not passed through the segment i_8-i_{10} at the same iteration of the outer loop.

However, not all the input data which cause execution of such a path will necessarily cause the error to show up. Clearly, in some cases the final values of matrix a will be correct, i.e., the matrix will be sorted and its elements will be a permutation of its initial values. This happens if either 1) the elements

[1]The loss of the initial value of $a(0)$ (i.e., 5) is caused by the fact that the sequence i_{11} through i_{13} does not actually swap $a(R1)$ and $a(R3)$ because $R0$ is taken as the value of $a(R3)$. Had i_{12} been written as $a(R1) := a(R3)$ the final value of the array would have been (8, 5, 7, 2, 1).

that are being interchanged by instructions i_{11} to i_{13} are equal (i.e., $a(R1) = a(R3)$) or 2) the value of $R3$ is equal to the current value of $R1$. The latter case might occur if $a(R1 + 1)$ was the largest element of subarray $a(R1:N)$ examined during the previous execution of the inner loop (cf. [2]).

A path whose execution might lead to eventual detection of an error will be referred to as an *error-sensitive* path. If an error is always detected along such a path the latter is called an *error-revealing* path. In the case of our program no error-revealing path exists, though the error can be detected if an error-sensitive path is traversed and the program input data [i.e., $a(0:N)$] does not satisfy the above error-masking conditions 1) and 2).

A testing technique is called *reliable* for an error if its application guarantees that the error will always be detected (cf. [1], [2]). In light of this definition a reliable testing strategy must lead to exercising at least one error-revealing path. In the case of our example program no error-revealing path exists so there is no reliable testing strategy for this program. Therefore, we adopt a weaker definition of a *viable* or *weakly reliable* strategy which requires that at least one error-sensitive path in the program be traversed. In the next section we show application of some of the control-oriented strategies to our example program.

III. Control Flow Oriented Strategies

Path Testing [1]: Each executable path through a program is tested at least once.

For the example program this strategy is viable because there exist paths which are error-sensitive. For example, for the initial array $a = (5, 7, 8, 1, 2)$ an error-sensitive path is executed because upon termination of the program an incorrectly sorted array $a = (8, 7, 7, 2, 1)$ is returned. There is a finite number of paths in the program that increase as N, the size of the array, increases. However, even for a reasonably small N, the number of paths grows so rapidly that the path testing strategy becomes impractical.

Branch Testing [16]: Each branch in the program is tested at least once. The strategy is not viable because there are paths that are not error-sensitive though they satisfy the requirement of the strategy. For example, for initial array $a = (8, 2, 4, 0, 1)$ there is a path such that every branch is executed and the output array is sorted correctly i.e., on termination $a = (8, 4, 2, 1, 0)$. It can be shown that for any input data which causes execution of this path, the output array is sorted correctly.

Boundary-Interior Testing [4]: A boundary test of a loop is a test which causes the loop to be entered but not iterated. An interior test causes a loop to be entered and then iterated at least once. For the sorting program there are only interior tests of the loops unless $N = 0$. The strategy is not weakly reliable because it might, like branch testing, generate an error-insensitive path.

IV. Data Flow Modeling

Let i be a simple instruction in a program with $x_1, x_2, \cdots, x_k$ as its arguments, i.e., those variables whose values are used by i to carry out the computation specified by i. i can be either an assignment instruction (e.g., $y{:=}F(x_i, \cdots, x_k)$), a test instruction (e.g., **if** $p(x_1, \cdots, x_k)$), an output data instruction (e.g., write $(x_i, \cdots, x_k)$) or a procedure call, providing that the input variables of that call are uniquely identifiable. "Use" of a variable occurs therefore in an instruction in which its value is first used before being eventually changed.

Let $d(x) = [x^1, x^2, \cdots]$ be set of *definitions* of variable x in the program. A definition of a variable occurs in an instruction which assigns a value to that variable. It can be an assignment instruction, an input data instruction or a procedure call. It should be noted that a procedure call instruction is treated as a black box with a set of input variables and a set of output variables; it is also assumed that during each execution of a procedure call all input variables are used and all output variables are defined (in fact it might not be true). We say that definition x^k of variable x is *live* at i if there exists a control path from x^k to i along which x is not redefined (cf. [10]-[15]).

For example, for the program schemata in Fig. 2 where subsequent definitions of variables are marked with superscripts, definition y^2 is live at i_4 and i_6 while y^1 is not because it is overwritten by y^2.

If x^k is live at i then x^k can possibly be used by i as an input value. Clearly, the requirement that a control path exists from x^k to i is of purely structural nature; in fact under normal operating conditions that path might not be *feasible*, i.e., it might not be executable. Determining path feasibility is therefore a semantic issue and cannot be done through static text analysis only.

The set of all live definitions of all input variables of instruction i will be referred to as its *data environment* and denoted by $DE(i)$.

For example, instruction i_{13} in the sorting porgram has the following data environment:

$$DE(i_{13}) = [R2^3, R3^1, R3^2].$$

The above notion of data environment is a simple model of the data flow in a program. It expresses the relationships between definitions and uses of the variables taken separately (cf. the notion of use-definition chain in [11]).

However, execution of an instruction with n arguments, $n \geqslant 1$, involves simultaneous use of an n-tuple of definitions of input variables from the data environment. This fact is formally expressed by the following notion of data context, which is a more complete model of the data flow in a program. By an *elementary data context* of an instruction i that uses a set of variables $X(i) = [x_1, x_2, \cdots, x_n]$ we mean a set $ec(i) = [x_1^{i1}, x_2^{i2}, \cdots, x_n^{in}]$ of definitions of all variables from $X(i)$ such that there exists a control path from the beginning of the program to i and all the definitions from $ec(i)$ are live at i when the path reaches i.

In other words, an elementary context of an instruction is a tuple of definitions of all arguments of that instruction that can possibly be used if a particular control path is executed; such a path will be referred to as a *testing* or *activating* path of the context.

By the *data context* $DC(i)$ of instruction i we mean the set of all its elementary contexts.

The above definition of an elementary context does not in-

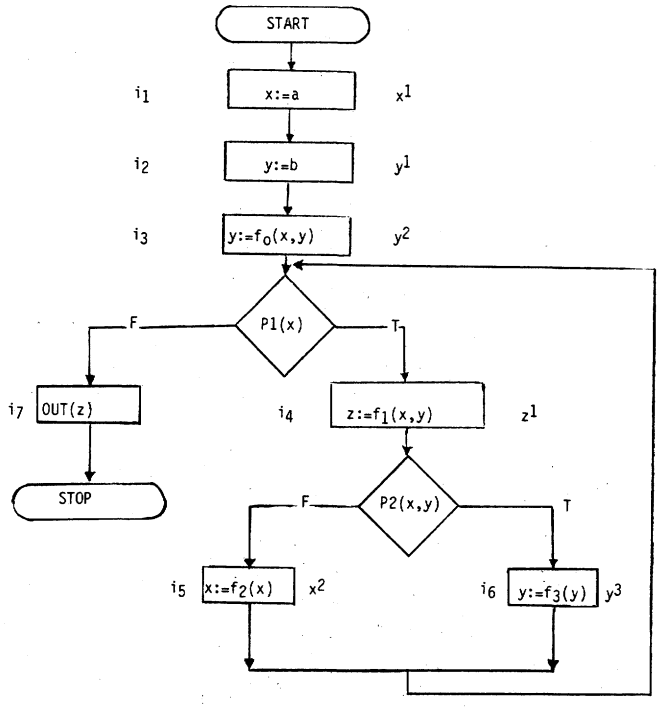


Fig. 2. A program schemata.

volve any ordering of the data definitions appearing in that context.

However, every activating path of a elementary context does impose such an ordering; clearly this is the order in which those definitions appear along that activating path. This gives rise to the notion of the *ordered* elementary *data context*. The latter is an ordered sequence of definitions from $ec(i)$ such that there exists a control path from the program beginning to i along which those definitions are activated in the order specified by that sequence and are not overwritten when the control path reaches i.

Consider, for example, the program in Fig. 2 for which

$$DC(i_4) = [(x^1, y^2), (x^1, y^3), (x^2, y^2), (x^2, y^3)].$$

The following is the set $ODC(i_4)$ of the ordered elementary contexts of i_4:

$$ODC(i_4) = [(x^1, y^2), (x^1, y^3), (y^2, x^2), (x^2, y^3), (y^3, x^2)].$$

It is clear that two elementary contexts from $ODC(i)$ might differ only in ordering of otherwise identical definitions, as in the case of (x^2, y^3) and (y^3, x^2) from $ODC(i_4)$.

As far as the sorting program is concerned there exists only one possible order of each elementary data context.

The following is the list of the data contexts of all but these instructions in the program which have no input variables:

$$DC(i_3) = [(R1^1, N^1), (R1^2, N^1)]$$
$$DC(i_4) = [(a^1, R1^1), (a^3, R1^2)]$$
$$DC(i_5) = [(R1^1), (R1^2)]$$
$$DC(i_6) = [(R2^1, N^1), (R2^2, N^1)]$$
$$DC(i_7) = [(a^1, R0^1, R2^1), (a^1, R0^1, R2^2), (a^1, R0^2, R2^2), (a^3, R0^1, R2^1), (a^3, R0^1, R2^2), (a^3, R0^2, R2^2)]$$
$$DC(i_8) = [(a^1, R2^1), (a^1, R2^2), (a^3, R2^1), (a^3, R2^2)]$$
$$DC(i_9) = [(R2^1), (R2^2)]$$
$$DC(i_{10}) = DC(i_9)$$
$$DC(i_{11}) = [(a^1, R1^1), (a^3, R1^2)]$$
$$DC(i_{12}) = [(R1^1, R0^1), (R1^1, R0^2), (R1^2, R0^1), (R1^2, R0^2)]$$
$$DC(i_{13}) = [(R3^1, R2^3), (R3^2, R2^3)]$$
$$DC(i_{14}) = [(R1^1), (R1^2)].$$

While deriving these contexts it has been assumed that an operation on an array element involves both the array variable as a whole and the index variable as well.

The above notions of the data environment and data context have been defined with respect to the statement level of a program. However, they might be also extended to the block level. To define the data environment and data context for a block one has to determine the input and output variables of a block.

A program variable is an *input* variable of a block if it is referenced within the block before it is assigned a new value in the block (if any). A variable is an *output* variable of a block if it is defined within the block. A block consisting of a single conditional is assumed to have input variables only.

A block can be viewed as a multiargument, multiresult instruction. A single variable in a block can be defined several times by the instructions contributing to the block. Therefore, by definition of an output variable of a block we shall mean the cumulative effect of all definitions of that variable within the block rather than a single (e.g., the last) definition in the block. For example, the instruction-level definitions a^2 and a^3 in our sorting program are replaced by the single definition a^2 at the block level.

With this interpretation of a block-oriented definition of a variable we can apply the notions of liveness, data environment and data context to the block level rather than to the statement level.

A block definition x^k of a variable x in block bi is *live* at block bj if there exists a control path from bi to bj along which x is not redefined. The set of all live definitions of all input variables of block bj is referred to as its *data environment* and denoted by $DE(bj)$. For example, block $b5$ in the sorting program has the following data environment:

$$DE(b5) = [a^1, a^2, R1^1, R1^2, R0^1, R0^2, R3^1, R3^2]$$

whose elements, with the exception of a^2, incidently coincide with the instruction-level definitions of the variables involved.

By an *elementary data context* of block b with $X(b) = [x_1, x_2, \cdots, x_n]$ as the set of its input variables we mean a set $ec(b) = [x_1^{i1}, x_2^{i2}, \cdots, x_n^{in}]$ of definitions of all variables from $X(b)$ such that there exists a control path from the beginning of the program to b and all the definitions from $ec(b)$ are live at b when that path reaches b.

By the *data context* $DC(b)$ of block b we mean the set of all it elementary data contexts.

The following is the list of data contexts of all blocks of the sorting program except those which have no input variables:

$$DC(\alpha) = DC(i_3)$$
$$DC(b2) = [(a^1, R1^1), (a^3, R1^2)]$$
$$DC(\beta) = DC(i_6)$$

$$DC(\gamma) = DC(i_7)$$
$$DC(b3) = [(a^1, R2^1), (a^1, R2^2), (a^3, R2^1), (a^3, R2^2)]$$
$$DC(b4) = DC(i_{10})$$
$$DC(b5) = [(a^1, R0^1, R3^1, R1^1), (a^1, R0^2, R3^2, R1^1), (a^2, R0^1, R3^1, R1^2), (a^2, R0^1, R3^2, R1^2), (a^2, R0^2, R3^2, R1^2)].$$

One can also define an *ordered* elementary *data context* of a block analogously to the ordered data context of an instruction.

V. Data Flow Based Testing Strategies

The notions of data environment and data context introduced in the previous section are particular models of data flow in a program. They capture the different relationships between the data definitions and their uses. The data environment and the data context can be used to guide the testing procedure. Depending upon the choice of data flow model one can obtain different testing strategies. A testing strategy would then aim towards exercising some control paths along which the elements of a chosen data flow model are "activated." In the following, two such strategies are considered. The first refers to the data environment while the latter to the data context. These strategies can be applied either to the instruction level or the block level of a program.

Strategy I

This strategy requires that liveness of each definition from the data environment of every instruction (block) be tested at least once. By the liveness of the definition at an instruction (block) we mean the fact that the definition is live at the instruction (block) during execution of the program.

Strategy II

This strategy requires that each elementary data context of every instruction (block) be tested at least once.

One can also consider a modified version of Strategy II which requires that each ordered elementary data context of every instruction (block) be tested at least once.

These strategies do not impose other restrictions on the choice of control paths which activate given definition of context. A set of paths which satisfy the requirements of a strategy is said to be a *testing set* for that strategy. Usually, no unique testing set can be determined by these strategies unless some additional criteria, for example, the shortest path requirement, is assumed.

As it happens in control oriented testing, the data oriented testing strategy is also of purely structural character. Clearly the aim of such a strategy is to exercise those paths in the program which activate certain use-definition chains in the data flow. The data environment strategy (Strategy I) employs for that purpose chains involving single variables only, while the data context approach (Strategy II) deals with chains involving vectors of simultaneously used variables. However, in both cases the uses in question are of potential character, because for some of them their testing paths might not be feasible.

It is therefore the programmer's responsibility to provide input data which cause the elements of a testing path set to be exercised. A path that is found infeasible might be so due to the design of the program and might not necessarily be symptomatic of error. In contrast, a feasible path might not be so in a correct design and its feasibility might indicate presence of an error. In any case, however, selection of input data for the testing set of a testing strategy falls in the category of semantic program analysis and its automatization still seems to be a distant goal, although some significant steps in this direction have been recently explored (cf. symbolic execution [1], [2]).

In what follows we briefly discuss viability of the two data oriented testing strategies when applied to our sorting program. Because the number of testing paths is quite large in both cases we do not embark on finding all of them but instead we concentrate on the possibility of finding an error-sensitive one among them.

Strategy I

This strategy is not viable for the sorting program because it does not guarantee execution of an error-sensitive path. However, because it seems to be the simplest application of data flow to testing it is of interest to ask when it might be reliable. An answer to this question is illustrated by the following fragments of a program in which two definitions of variable x, x^1 and x^2, can be used by the same instruction i:

```
      .
      .
      .
if p then
x := 0          'x1'
else
   x := f(x)    'x2'
end if
      .
      .
      .
if q then
      .
      .
      .
i: y := g(a, y)/x
end if
```

If definition x^1 is live at i then the "zero division" error occurs. Strategy I is viable for this error by forcing the liveness preserving path to be executed. Control flow oriented strategies might not guarantee detection of the error except for the path testing strategy.

Strategy II

Unlike the data environment strategy this strategy involves testing simultaneous liveness of all variables contributing to the input set of an instruction or block. It is weakly reliable for the error when applied to the block level. To see this, consider the following elementary data context of block $b5$,

$$ec(b5) = (R3^1, a^2, R1^2, R0^1)$$

which appears in its "natural" control order (note that other orders of $ec(b5)$ as say, $(R0^1, a^2, R3^1, R1^2)$ are infeasible). To activate $ec(b5)$ one has to provide an input array whose dimension is equal to or greater than 3 and whose two first

entries, at least, are initially ordered. Clearly, 1) $R3^1$ and $R0^1$ require that $b3$ has not been executed when control reaches $b5$, which is equivalent to stating that $a[0:R1]$ be initially ordered and 2) $R1^2$ and a^2 require that the outer loop be executed at least twice which means that a has to contain at least three elements, or $N \geqslant 2$.

In such a case during the second execution of the outer loop, when $ec(b5)$ is activated (i.e., control reaches $b5$), there is $R3 = 0$, $R1 = 1$, and $R0 = a(R1)$. Next the swapping routine copies $a(R1)$ into $a(R3)$, overwriting the initial value of $a(0)$. It should be observed that the associated operation of assigning $R0$ to $a(R1)$ does not result in error because $R0 = a(R1)$. Thus any testing path which activates $ec(b5)$ is error-sensitive. However, if initially $a(0) = a(1)$ then the error will not show up because error revealing is guaranteed for no structural testing of the program unless supplemented by another data selection criterion. For example, the error will be detected for any distinct entries of $a[0:N]$ which lead to activation of $ec(b5)$.

To illustrate this suppose that $N = 2$ and the initial value of a is (8, 7, 2). During the second traversal of the outer loop $b5$ is entered with $R3 = 0$, $R1 = 1$ and exited with $R1 = 2$ and new value of $a = (7, 7, 2)$. The latter wll show up as an error on termination of the program.

However, it should be noted that if the missing reinitialization $R3 := R1$ were present in the program, the context $ec(b5) = (R3^1, a^2, R1^2, R0^1)$ would not have been feasible because $R3^1$ would have been overwritten by the assignment $R3 := R1$.

VI. Reliability of the Approach

Although it has been shown that data flow testing proved more powerful than its control oriented counterparts when applied to two sample programs, it would be premature to claim its superiority. A thorough study is needed to arrive at sound conclusions about the strengths and weaknesses of the approach. Such an analysis should take into account 1) the potential ability of the method to detect errors of certain kind and its failure to detect other errors and 2) the complexity of the method measured by the number of tests required.

None of those questions seems to be well-defined in the absence of a language independent model of errors and the way they affect the observable behaviour of computer programs. The known works in this direction [16], [22], although providing a deeper insight into the causes of errors at different stages of program development (specification, design, coding, verification) do not provide a general language independent framework in which different types of errors could be defined and their propagation studied. An attempt to arrive at such a framework has been recently undertaken [8] on the basis of a program model developed in [20] and although it calls for a further refinement we will use the classification of errors put forward there in the following short informal discussion of data flow testing reliability.

Programming errors and their impact on the operational behavior of the program should be studied with respect to a given level of program decomposition. As far as "pure testing" is concerned, when debugging is the objective of a separate stage of the program development process, it is the highest, whole-program level that is of interest. At that level the program is viewed as a black box that transforms input data into output data.

Correctness of the transformation is assessed on the basis of a specification of the program that, whether formal or informal, serves as the ultimate "test oracle." Assuming that the specification determines the total correctness of the program, incorrectness of the program is manifested either as nontermination or as incorrect results of an otherwise terminating program.

Therefore, nontermination or incorrect results are the only observable, run-time effects of any static, textual error that affects the program text. To be able to ask which textual errors can be revealed by a particular testing strategy, a program decomposition level more detailed than the black-box model is required. To this point we assume that the program can be represented as a system of interconnected modules with explicitly defined data and control flows (cf. [20]). This gives rise to the classification of textual errors as either transformation or control errors. Transformation errors account for incorrect transformations of data by the modules involved while control errors account for incorrect control decisions and missing path cases.

It should be noted, however, that the above classification is to be considered with respect to a given level of program decomposition. A transformation error, for example, can be caused by a number of transformation and/or control errors at a lower level of decomposition. As a rule both transformation and control static errors at a given level manifest themselves at the higher levels either as a transformation error or as nontermination of a higher level module.

Which of the above classes of errors can be detected by data flow testing? Let us first make clear that no structural testing strategy will guarantee ultimate detection of an error if the program being tested does not contain an error-revealing path. For example, an error is guaranteed to show up for a single-path program (a sequence of instructions) only if the program works incorrectly for all inputs. Clearly, the single path of such a program is error-sensitive but test data which reveal the error must be chosen on other than structural grounds. Therefore, for an arbitrary program, no structural oriented testing is reliable and the power of a structural testing strategy should be formulated in terms of its viability rather than its reliability.

Consider now a transformation error in a block which results in incorrect values of some of the block output variables; the corresponding block-level definitions of those variables (representing the cumulative effect of all assignments to them within the block) will be referred to as the incorrect definitions. Data flow testing requires that every instruction which uses those definitions be activated at least once for every possible combination of the incorrect definitions and the other definitions used by that instruction. Therefore, an error-sensitive path will ultimately be traversed because all possible uses of the incorrect definitions will be exhausted.

If a definition can be returned as the final (output) value of the variable involved, i.e., when there is a definition-clear path

from the definition to the program end, then the corresponding stop (exit, return) instruction can be thought of as using the definition. Consequently, data flow testing will require that the definition-clear path be exercised. For example, in the following binary search program computing the integer square root of a natural n [21]:

```
        integer function sqrt (n)
        begin integer n, l, n, t
        l := 0;
        u := n;
init:   t := (l+u)/2;          /* t^1 */
more:   if t*t>n then u := t else l := t fi;
        if (t=(l+u)/2) then
i:      return (t)             /* DC(i) = [t^1, t^2] */
        else t := (l+u)/2 fi;  /* t^2 */
        goto more;
        end sqrt
```

there is an error in instruction init which shows up only for $n = 1$. No control testing (except for path testing) guarantees detection of the error, but data flow testing requires that definition t^1 of $DC(i)$ be activated, which can be accomplished only if $n = 1$.

A typical case of a transformation error is a missing assignment to a variable. One might assume that such an error cannot be reliably detected because of the missing definition-use chains which would have appeared in the absence of the error. A closer analysis, however, shows that this is not always the case. Clearly, a missing assignment in a block gives rise to definition-use chains which would not have existed had the assignment been present. Those chains are error sensitive when the error does not affect functionality of the block, i.e., when there is another assignment to the variable involved within the block (by the functionality we mean the input and output variables of a block).

A transformation error may, however, affect the functionality of a block and then some new error-generated contexts may not be error sensitive. This is the case of our sorting program. Suppose that instruction $R3 := R1$ is present in the latter; then the correct intended data context of block $b5$ is the following:

$$DC(b5) = [(a^1, R0^1, R3^m, R1^1), (a^1, R0^1, R3^2, R1^1), (a^2, R0^1, R3^m, R1^2), (a^2, R0^2, R3^2, R1^2)]$$

when $R3^m$ denotes the "missing" definition of $R3$. Note the definition $R3^1$ does not appear in the above.

When $R3^m$ is missing those elementary contexts which contain $R3^m$ disappear and are replaced by new contexts that otherwise would have been infeasible:

$$(a^1, R0^1, R3^1, R1^1,), (a^2, R0^1, R3^2, R1^2), (a^2, R0^1, R3^1, R1^2).$$

Although all of the above are incorrect contexts, only the last two are viable, i.e., only their activation leads to execution of error-sensitive paths. The first context $(a^1, R0^1, R3^1, R1^1)$ is not viable because $R3 = R1$ when it is activated.

Similar reasoning can be applied to the control error case; it seems that data flow testing is not viable for control errors unless certain conditions are met. For example, if in the instruction **if** $x \leqslant y$ **then** $P1$ **else** $P2$ the predicate $x \leqslant y$ is misspelled as $x < y$, then one cannot expect that data flow testing will cause execution of $P2$ for $x = y$ unless some data contexts that appear in $P1$ or $P2$ will require that.

A missing path fault also seems to not be easily caught by data flow testing unless some unusual conditions occur. For example, if a missing path causes the lack of initialization of a variable then that data-flow anomaly will be detected by the method.

A general conclusion that might be drawn from the above discussion is that data flow testing is viable for transformation faults, but it is not viable for control errors. The latter drawback is, however, typical for control oriented strategies, too. Clearly, a missing path cannot be exercised by a control strategy nor can the latter guarantee detection of a subtle incorrect decision error as, for example, the misspelling error discussed above. A misspelling error, however, that swaps two branches in a decision statement (e.g., $x \leqslant y$ written as $x > y$) will be detected by either branch testing or by data flow testing.

VII. Conclusions

The proposed method of program testing involves data flow analysis in the program as its prerequisite. There exists a number of algorithms for the data flow analysis problems that emerge in compiler design [7], [9]-[15]. In most cases they focus on code optimization involving, for example, removal of useless code from the object program or sharing some machine memory locations among different variables. Although they are not directly applicable to the data flow testing strategies, it seems that they might readily be adapted for that purpose. For example, the live-dead analysis aiming towards determination of the live definitions that can reach an instruction [11] can be extended to generate paths along which those definitions are preserved.

Such testing oriented data flow analysis could be a part of a sophisticated debugger. Its output would then be the set of paths which should be tested according to the chosen data flow oriented strategy. Some optimization could also be carried out to arrive at a minimal set of control paths which cover all the testing paths for a given strategy. This task might be fully automated, although it would be a programmer's responsibility to provide input test cases that cause those paths to be executed.

In summary, the data flow strategy offers a new approach to testing whose full potential, however, has yet to be explored. Some interesting issues that seem worthwhile for further study are the following:

1) The use of a hierarchical, data oriented program decomposition for testing purposes. Such an approach would enable one to reveal some hypothetical subfunctions that contribute to the overall program design. In this respect, data oriented testing would have some flavor of specification-driven, functional testing (cf. [19]).

2) Comparative analysis with the control oriented approach, particularly with branch testing. To this point one can say

that data flow testing provides a finer test completeness measure than branch testing while avoiding the path explosion problem typical for path testing. Loops are exercised until the existing definition-use chains are exhausted and though, unlike in the control oriented structural testing, no finite limit on loop iterations appear, the actual number of iterations is always finite, owing to the finite number of chains to be tested.

3) Methods for combining data flow with functional specification-driven testing to enhance viability of the approach.

Let us stress again, however, that correct solutions to those problems should be formulated in terms of a sound, language independent model of programming errors in which their effect on program behavior could be studied.

Acknowledgment

The authors wish to express their gratitude to Prof. D. Boddy from the School of Engineering and Computer Science, Oakland University, and to Dr. J. Goodenough from the Editorial Board of this TRANSACTIONS for their careful reading and valuable comments which helped improve the final version of the paper. This applies both to the substance of the paper and overcoming the passive resistance of the English language. Thanks are due to anonymous reviewers whose suggestions have also been taken into account.

References

[1] W. E. Howden, "Reliability of the path analysis testing strategy," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 208–214, Sept. 1976.

[2] —, "Symbolic testing-design techniques, costs and effectiveness," Nat. Bureau Standards, Dep. Commerce, Washington, DC, NBS-GCR-77-89, May 1977.

[3] C. V. Ramamoorthy and S.B.F. Ho, "Testing large software with automated software evaluation system," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 46–58, Mar. 1975.

[4] W. E. Howden, "Methodology for the generation of test data," *IEEE Trans. Comput.*, vol. C-24, pp. 554–559, May 1975.

[5] J. C. Huang, "Detection of data flow anomaly through program instrumentation," *IEEE Trans. Software Eng.*, vol. SE-5, pp. 226–236, May 1979.

[6] L. J. Osterweil and L. D. Fosdick, "Data flow analysis as an aid in documentation, assertion generation, validation and error detection," Dep. Comput. Sci., Univ. Colorado, Boulder, Rep. CU-CS-055-74, Sept. 1974.

[7] L. D. Fosdick and L. J. Osterweil, "Data flow analysis in software reliability," *Comput. Surveys*, vol. 8, pp. 305–330, Sept. 1976.

[8] J. W. Laski, "A hierarchical approach to program debugging," *SIGPLAN Notices*, vol. 15, pp. 77–85, Jan. 1980.

[9] J. M. Barth, "A practical interprocedural data flow analysis algorithm," *Commun. Ass. Comput. Mach.*, vol. 21, pp. 724–736, Sept. 1978.

[10] M. S. Hecht, "A simple algorithm for global data flow analysis problems," *SIAM J. Comput.*, vol. 4, pp. 519–532, Dec. 1975.

[11] K. Kennedy, "A comparison of two algorithms for global data flow analysis," *SIAM J. Comput.*, vol. 5, pp. 158–180, Mar. 1976.

[12] F. E. Allen, *A Basis for Program Optimization, Information Processing 71.* North-Holland, 1972, pp. 385–390.

[13] M. S. Hecht and J. Ullman, "Characterizations of reducible flow graphs," *J. Ass. Comput. Mach.*, vol. 21, pp. 367–375, July 1974.

[14] B. K. Rosen, "High-level data flow analysis," *Commun. Ass. Comput. Mach.*, vol. 20, 10, pp. 712–724, Oct. 1977.

[15] F. E. Allen and J. Cocke, "A program data flow analysis procedure," *Commun. Ass. Comput. Mach.*, vol. 19, pp. 137–147, Mar. 1976.

[16] J. B. Goodenough and S. L. Gerhart, "Toward a theory of test data selection," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 156–176, 1975.

[17] E. J. Weyuker and T. J. Ostrand, "Theories of program testing and the application for revealing subdomains," *IEEE Trans. Software Eng.*, vol. SE-6, pp. 236–246, May 1980.

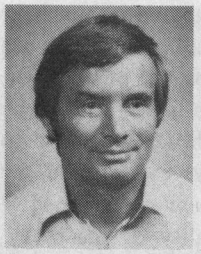
[18] L. J. White and E. I. Cohen, "A domain strategy for computer program testing," *IEEE Trans. Software Eng.*, vol. SE-6, pp. 247–257, May 1980.

[19] W. E. Howden, "Functional program testing," *IEEE Trans. Software Eng.*, vol. SE-6, pp. 162–169, Mar. 1980.

[20] J. W. Laski, *The Semantic Analysis of Modular Programs (An Approach Towards the Software Reliability Studies)*. Polish Sci. Pub. PWN, Warsaw, 1981, ISBN 83-01-01692-2.

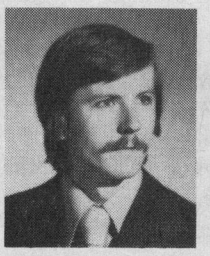
[21] R. G. Hamlet, "Testing programs with finite sets of data," *Comput. J.*, vol. 20, no. 3, pp. 232–237, 1977.

[22] S. L. Gerhart and L. Yelowitz, "Observations of fallibility in applications of modern programming methodologies," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 195–207, Sept. 1976.



Janusz W. Laski received the M.Sc. degree in telecommunications in 1962 and the Ph.D. degree in computer science in 1969, both from Technical University of Gdansk, Gdansk, Poland.

From 1962 to 1968 he was a Maintenance Engineer at a computer center and since 1968 he has been at Institute of Informatics, Technical University of Gdansk. Meanwhile has was a Consultant to the United Nations/FAO working on development of a real-time operating system for a shipborn installation (1969–1972). In 1972 he was a Visiting Lecturer in Polytechnic of Central London and in 1979 a Visiting Research Professor in the Department of System Design, University of Waterloo, Canada. Currently, he is an Associate Professor in the School of Engineering and Computer Science, Oakland University, Rochester, MI. His current research interests are software design and verification methods.



Bogdan Korel was born on September 1, 1950, in Rybnik, Poland. He received the M.S. degree in electrical engineering from Technical University of Kiev, U.S.S.R., in 1974.

Since 1974 he has been working in the Institute of Control Systems, Katowice, Poland. His major research interests are in the areas of software reliability and parallel processing. He is currently enrolled in the Ph.D. program at Oakland University, Rochester, MI.