Feasible Test Path Selection by Principal Slicing*

István Forgács[1] and Antonia Bertolino[2]

[1] Computer and Automation Institute, Hungarian Academy of Sciences, XI. Kende u. 13-17, Budapest, Hungary
[2] Istituto di Elaborazione della Informazione, Consiglio Nazionale delle Ricerche, via S. Maria, 46, 56126 Pisa, Italy

Abstract. We propose to improve current path-wise methods for automatic test data generation by using a new method named *principal slicing*. This method statically derives program slices with a near *minimum number of influencing predicates*, using both control and data flow information. Paths derived on principal slices to reach a certain program point are therefore very likely to be feasible. We discuss how our method improves on earlier proposed approaches, both static and dynamic. We also provide an algorithm for deriving principal slices. Then we illustrate the application of principal slicing to testing, considering a specific test criterion as an example, namely branch coverage. The example provided is an optimised method for automated branch testing: not only do we use principal slicing to obtain feasible test paths, but also we use the concept of spanning sets of branches to guide the selection of each next path, which prevents the generation of redundant tests.

Keywords: automatic test data generation, ddgraph, influencing predicates, PDG, principal definition, slicing

1 INTRODUCTION

The problem addressed in this paper is how to select some program path from the entry to a certain point (or to some certain points) of a given program module. This problem is common to all white–box test methods, specifically, to coverage criteria [2], both control flow– and data flow–based, to domain [19], and to fault-based (for example mutation) testing [6], in the unit test phase of software.

White–box approaches are more usefully applied to evaluate the adequacy of the exercised test set (e.g., wrt the achieved branch coverage in branch testing, or the mutation score in mutation testing), and to decide whether testing can be stopped. However, in a case which is quite common the test set is deemed inadequate, and so more tests have to be selected to improve test adequacy. In

* Reseach was supported in part by Hungarian National Foundation, grants 7314, 023307, by Soros Foundation, and in part by OLOS HCM Network under EC Contract No. CHRX-CT94-0577.

such a situation, the test criterion (e.g., branch coverage or mutation) is used to guide the selection of new tests. Generally speaking, each newly selected test will require the determination of a program path to be executed (e.g., to an as yet not covered branch in branch testing, or to the location of the syntactic change of an as yet not killed mutant). Determining a suitable path is the task we consider here.

The difficulty of this task is in how to determine a *feasible* path, i.e., a path that is executable under some input data. As is well known, feasible path selection is undecidable in general [18].

Dynamic techniques proceed by attempts and, when they succeed, provide directly the input data which exercise the selected path. In the *goal-oriented* approach [12], some "goal" nodes to be traversed in the program flowgraph are given. A test input is selected randomly, and the associated program path is analyzed. If this path excludes a goal node, the branch "responsible" for the undesired execution is identified. Function minimization search algorithms are then used to find new input data which alter the flow of control so that the "good" branch is followed instead. This iterative search either leads to an input under which all the goal nodes are traversed, or fails (i.e., no appropriate test data for these goal nodes are found). The *chaining approach* [7] is an extension of the goal-oriented approach, in which data flow information is also used. If the goal-oriented approach fails because of a given predicate p, a new sequence of nodes (event sequence) is generated for which the desired branch on p can be executed. Event sequences are generated so that the data flow (definition use pairs) of each new event sequence is different from earlier failed event sequences.

A completely different approach to overcome the path feasibility problem consists of selecting paths with a low number of predicates, through the use of static analysis techniques. This approach follows from the very intuitive concept that the shorter a path is, the more likely it is that it is feasible. This has also been empirically validated. Yates and Malevris [20] investigated 642 generated control paths. They showed that the probability of feasible paths in a program decays exponentially with the number of traversed predicates. For paths in which the number of traversed predicates is greater than five, the number of infeasible paths exceeds the number of feasible paths. Therefore, a good path selection method should select paths so that the number of traversed predicates is as low as possible.

Heuristics have been proposed to select paths according to this principle, e.g., in [3], [21]. However, all the methods proposed so far to find such "short" paths only consider control flow information, while data flow dependencies are ignored. Indeed, the testers' main aim is generally not to execute a specific path, but rather to reach one (or more) selected program point(s). In order to satisfy this goal, what actually should be minimised is not the total number of predicates in the entire path, but the number of predicates which are actually determinant to reach that(those) point(s). We call these determinant predicates *influencing predicates.* Those predicates which do not affect the selected predicate, the *non-influencing predicates*, need not be considered for path selection.

Consequently, we propose a new approach for selecting feasible test paths. This approach consists of selecting those paths which reach a specified program point with a number of influencing predicates that is as low as possible. To do this, we introduce a new method called *principal slicing*, which supplies a program slice to a specified program instruction with an almost *minimum number of influencing predicates.*

A slice [17] [13] is a program segment which contains all and only those statements that might affect a set of specified variables at a given program point. Intuitively, in our new method principal slices are generated so that an appropriately derived set of program statements is traversed during program execution. In this way we can get a program segment which is smaller than the slice which would be obtained without having to traverse the appropriate set of statements, and yet still yields the property of a slice. Assuming that the appropriate statements are executed, the principal slice contains all and only those statements that might affect a set of variables referenced at a given program point. All other program parts can be removed, thus reducing the number of predicates to be then considered during the phase of input data generation.

Our method improves on earlier static methods, in that these, being based exclusively on control flow analysis, only select the shortest route on the entire program flowgraph. By contrast, based on data flow analysis, we first obtain a principal slice which may contain far fewer predicates than the original program and then we can more easily derive a test path on the flowgraph of this reduced module.

Finally, our method also improves on dynamic methods for path selection, in that principal slicing explicitly includes specific techniques for finding a set of flowgraph nodes such that a path traversing such nodes is very likely to be feasible. In contrast, the goal-oriented approach and its extension of the chaining approach select arbitrary program paths with respect to the influencing predicates. Our method is thus more comprehensive.

In the next section we provide the necessary background. In Section 3, after a brief introduction to (static and dynamic) slicing techniques, we describe the method of principal slicing, and provide an algorithm to derive principal slices. Then, in Section 4, an example of an application of our method to branch testing is given. In this example, we combine principal slicing with the concept of spanning sets of entities for coverage testing criteria [14]. Spanning sets help to reduce and estimate the number of test cases needed to satisfy a given coverage criterion. Finally, the conclusions of our investigation are drawn in Section 5.

2 BACKGROUND

The control flow of a program module M is typically represented on a digraph $G(M)$, called a flowgraph. A digraph $G = (N, A)$ consists of a set N of nodes and a set A of arcs, where an arc $e = (T(e), H(e))$ is an ordered pair of *adjacent* nodes, called *Tail* and *Head* of e, respectively. What changes from one author's flowgraph to another's is the mapping from program elements (instructions and predicates) to flowgraph elements (arcs and nodes). We shall use a flowgraph model called a *ddgraph* [3]. A program basic block (i.e., an uninterruptible sequence of program statements) is mapped onto a ddgraph arc, while a divergence or a junction in the program control flow is mapped onto a ddgraph node (with outdegree or indegree ≥ 2, respectively). In some cases, an arc is introduced that does not correspond to a program block, but nevertheless represents a possible course of the program control flow (e.g., the implicit *else* part of an *if* statement). A program branch corresponds to a ddgraph arc leaving a node with outdegree ≥ 2. Note that, by construction, a ddgraph does not contain *procedure nodes*, i.e., nodes with just one arc entering *and* just one arc leaving them. Formally, a *ddgraph* $G = (N, A, e_0, e_k)$ consists of a digraph with two distinguished arcs e_0 and e_k, which are the unique entry arc and the unique exit arc, respectively. For each node $n \in N$, except $T(e_0)$ and $H(e_k)$, $(indegree(n) + outdegree(n)) > 2$ (while $indegree(T(e_0)) = 0$ and $outdegree(T(e_0)) = 1$, $indegree(H(e_k)) = 1$ and $outdegree(H(e_k)) = 0$).

As an example, ddgraph $G(Example)$, corresponding to the simple program *Example* in Figure 1, is shown in Figure 2.

A path $p = (e_0, e_1 ..., e_k)$ in a ddgraph is a finite sequence of adjacent arcs. A path is simple if all its nodes are distinct. A *control path* is any path in $G(M)$, while a *program path* is a path that has been actually executed on M for a given input.

Considering a module M, the use of a variable v in an instruction u and a definition for v in an instruction d form a *du pair* if the value of v defined in d can potentially be used in u. For this case we say that u is *directly data dependent* on d. Instruction u is *data dependent* on instruction d if there is a sequence $I_1, I_2, ..., I_k$ $(k > 1)$ of instructions, such that $d = I_1$, $u = I_k$, and I_{i+1} is directly data dependent on I_i for $1 \leq i \leq k - 1$. Another type of dependence is control dependence. An instruction I_p is *control dependent* on an instruction I_r if every program path beginning with entry and ending with I_p traverses I_r, I_r has at least two successors, and there is a control path through I_r that does not include I_p.

The *program dependence graph* for module M, denoted by PDG, is a digraph whose arcs are labelled. The nodes of PDG represent individual program instructions in M; in addition, there is an "enter" node. Some other nodes may be added to PDG (see [8] for details). An arc in the PDG represents either a control or a data dependence. The source of a control dependence arc is the enter node or a predicate node. Control arcs are labelled according to the result of the predicate. For example, the two branches of an *if* statement are labelled with *True* (*then* part) or *False* (*else* part). For *case* predicates, a different number is attached to each branch. Examples of PDGs can be seen in Figure 5.

In this paper we investigate intraprocedural dependencies, i.e., modules with no procedure calls. The modules are structured programs. Data types are not limited, thus array and pointer types are not excluded.

```
procedure Example
  read(y,z)                 (0)
  x = z * 2                 (1)
  if z > 0 then             (a)
      w = z                 (2)
  else
      w = y                 (3)
  endif
  if y > 0 then             (b)
      x = 10                (4)
  else
      x = x - 10            (5)
      while w < y do        (c)
         w = w + 2          (6)
      endwhile
  endif
  if z + y > 10 then        (d)
    x = y + 2               (7)
  else
    y = w + 2               (8)
    x = x + w * 2           (9)
  endif
  if x = 0 then             (e)
    write(y)               (10)
  endif
endprocedure
```

Fig. 1. A program module

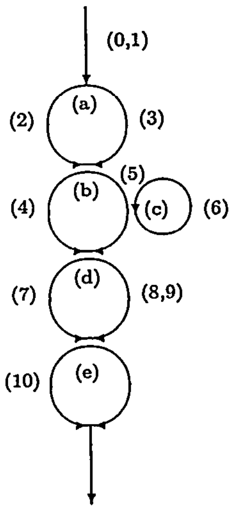

Fig. 2. A ddgraph

3 PRINCIPAL SLICING

3.1 Slicing Overview

Program (or *static*) slicing, originally defined by Weiser in [17], is a method for decomposing programs into segments. In Weiser's concept, a slice consists of an executable program segment involving all statements and predicates that might affect a set V of variables at a program instruction I. The pair $C_s = (I, V)$ is said to be a *slicing criterion.* In some cases [8], it is assumed that the variables in the set V are referenced at program instruction I.

A different slicing method is called *dynamic* slicing [13]. A slicing criterion for dynamic slicing is a triple $C_d = (x, I^q, V)$, in which x is the selected input, I^q denotes an instruction I at an execution position q, and V is a set of variables (because an instruction I may be executed several times, the qth time instruction I is executed is referred to as the qth execution position of I and referred to as I^q). Intuitively, a dynamic slice is an executable program segment whose behavior is identical to the original program with respect to a given execution position for a specified input.

Static and dynamic slicing differ in that a dynamic slicing criterion is defined with respect to a given input x, i.e., the program M is actually executed along a program path associated with x. In contrast, static slicing assumes arbitrary program paths in a reduced sub-program of M. A second difference is that the static slicing criterion requires a program instruction I, while dynamic slicing entails specifying a given execution point of that instruction. Dynamic slicing leads to smaller slices than static slicing [10].

Slicing has many applications in software engineering, e.g., algorithmic debugging, differencing and integration, testing, reverse engineering, and maintenance. An extensive overview of program slicing techniques and applications is provided in [10].

3.2 Intuition for Principal Slicing

In this paper we use slicing to improve automatic test data generation for path-wise test strategies. Specifically, we consider the problem of selecting a suitable program path reaching a certain point of a given program module.

For the following discussion, without loss of generality, let the program point to be reached be q, where q is a predicate from which a branch to be traversed forks. We also refer to this predicate as the *current predicate.*

We observe that those predicates which have no effect on the current predicate q can be neglected. A predicate p has no effect on another predicate q if under any input the result (*true*, $false$) of q is the same for both for the original program and the one which whould be obtained by removing p and all the statements (transitively) control dependendent on p. Depending on whether they affect the current predicate or not, predicates are referred to either as *influencing predicates* or as *non-influencing predicates* with respect to q.

If we apply static slicing the influencing predicates are unambiguously determined. However, considering dynamic slicing, the influencing predicates strongly depend on the program path actually executed. In fact, in a chosen program path a definition may be redefined without it having been used or may not be traversed at all, and thus it has no influence on q. If such a definition was control dependent on a predicate, then q may also be independent of this predicate. Thus, this non-influencing predicate can be ignored when an input has to be determined for the chosen program path although it may contain the non-influencing predicate. Therefore, instead of searching for a minimum number of predicates from the entry to a given predicate q, we should determine *the minimum number of influencing predicates* between these two nodes.

The question now is: how can we construct a slice that contains as few influencing predicates as possible? By applying static slicing, we would obtain a very inefficient result, since (as explained above) a program path might contain far fewer influencing predicates than a path selected arbitrarily from the static slice. Applying dynamic slicing is inefficient as well, because again we may find dynamic slices that contain many more influencing predicates than others. In addition, our final objective is precisely to find an input which executes a selected program instruction. Assuming that a priori we have such an input (as would be required by the dynamic slicing criterion) is a tautology. In conclusion, we would need to develop a method that is static, yet which finds and removes as many non-influencing predicates as possible.

This paper introduces a new slicing method, referred to as *principal slicing*, which results in a slice with an almost minimum number of influencing predicates. As we will show, one way to do this is to find a suitable set of *principal definitions* in the program to be traversed at any program execution. A principal definition can be any definition that, if executed, reduces the number of influencing predicates. Traversing *the principal instructions* containing these principal definitions ensures that (hopefully many) predicates have no influence on the current predicate. Thus, if we determine these principal definitions effectively, we can get a slice containing an almost minimum number of influencing predicates.

For instance, let us consider the module in Fig. 1. Let the current predicate be (e). Then, the definition of variable x in (7) is a principal definition that is worth considering. In fact, if, in order to reach (e), we select to traverse (7), then only predicate (d) is influencing wrt current predicate (e) since predicates (a), (b) and (c) do not affect it. On the other hand if we select the *else* branch from (d), then there are at least three influencing predicates: (a), (b) and (d). We will show in the next section how principal definitions can be found using data flow analysis.

By comparison, the static slice for C_s=((e), {x}) would contain the whole procedure except instructions (8) and (10); thus all the predicates are influencing.

Hence, principal slicing consists of two steps: i) determining an effective set of principal instructions Q; and ii) reducing the program module M to a slice M' on which we select program paths traversing all the instructions in Q. The method is static, in that we determine the principal instructions and module M' prior to program execution.

Definition *Principal Slicing*

A principal slicing criterion is a triple $C_p = (Q, I, V)$, in which Q is a set of principal instructions to be traversed, I is the target instruction and V is the set of variables that are referenced at the instruction I. A principal slice of a program module M on slicing criterion C_p is any syntactically correct and executable module M' obtained by removing zero or more statements from M, so that M' contains all the statements that might affect the elements of V at I, provided that any program path selected from M' always traverses all the elements in Q. The expression "always traverses" means that if $r \in Q$ is within a loop, then r is traversed for each iteration of the loop.

Note that for principal slicing, V is a set of variables referenced at I as in [8]. This requirement is sufficient for our purposes, and makes slice determination easier. Also note that there may be control paths that exclude an $r \in Q$, but we only select program paths that include *all* the elements in Q.

Let us compare principal slicing to both static and dynamic slicing. The main difference between principal and static slicing is that principal slicing requires some instructions to be traversed during the program execution, while static slicing does not. Both require static analysis of programs. Dynamic slicing differs from principal slicing as follows: (a) dynamic slicing requires program execution prior to slicing, while principal slicing does not; (b) in this way, dynamic slices contain instructions along one program path, instead of a set of possible paths as principal slices do; (c) a principal slicing criterion requires an instruction at which the original program should be sliced, while dynamic slicing requires an execution position of a given instruction. Based on this comparison, one may think that dynamic slicing leads to smaller slices (i.e., with fewer predicates) than principal slicing. This is not so. Obviously, an optimal algorithm for principal slice determination does not exist, therefore dynamic slices containing fewer predicates may exist. Other dynamic slices, however, might contain more predicates. Since the goal of principal slicing is to find principal definitions which lead to slices with an (almost) minimum number of influencing predicates, we believe that principal slices contain fewer influencing predicates than an average dynamic slice.

3.3 An Algorithm for Principal Slice Determination

In this section we give an algorithm for determining principal slices. The key part of this algorithm is the determination of Q, i.e., of the principal instructions to be traversed.

Let us recall that we are assuming that the instruction in C_p is q, or, the current predicate. A first step is to search for (in order to remove them) those predicates that do not affect q for arbitrary inputs. This can be obtained by applying "standard" techniques of static slicing [17], [15]. When the static slice has been determined, the *PDG* of the sliced program is constructed as a starting point for the subsequent analysis. In the remainder of this section, *PDG* will refer to the procedure dependence graph of the sliced module (rather than of the original).

The central issue is what other slice simplifications are possible. If we select a priori a set of "good" paths traversing the current predicate, then some more predicates can be removed. Indeed, along some paths the effect of some definitions does not arise. This can happen in two cases.

- Case 1: a definition is redefined and thus does not affect q, although there may exist another path along which this definition (transitively) influences the predicate.
- Case 2: a definition is not traversed by the selected path.

Let us first concentrate on Case 1. If a definition d_2^x of variable x always redefines another definition d_1^x of the same variable, d_1^x can be ignored. We refer to the latter as a *substitutable definition.* For a definition being substitutable, we have both static and dynamic conditions to be satisfied, i.e., the first condition is applied during the (static) derivation of the principal slice, while the second is considered for the selection of the test input. Case 1 is satisfied when both static and dynamic conditions hold. The static condition is that whenever d_2^x is traversed, it prevents any possible effect of d_1^x on q, that is to say traversing d_2^x should avoid any possible use of definition d_1^x in an assignment or a predicate statement (a write statement, for instance, would produce no effect). This is a necessary condition in order to exclude any program path in which u^x actually uses the value defined by d_1^x.

To check whether this condition holds, we use flow analysis after PDG reduction. For each definition d_p^x of every variable x in the sliced module, we consider the corresponding node m in PDG and take the node n from which there is a control arc to m in the PDG. Since every execution should traverse the principal definitions, non-executable nodes and arcs can be deleted. First, all the control arcs leaving n and having a different label than the one entering m can be deleted from the PDG. We repeatedly go backward along control arcs analysing each traversed node n_i in similar way, i.e., all outgoing control arcs with different labels than the one by which n_i is reached can be deleted from the PDG. Next, during a forward process the nodes (and their incoming and outgoing arcs) with no entering control arcs are also removed. This process is repeated, i.e., each node for which the entering control arcs is missing is deleted with all its entering and leaving arcs. In this analysis, we mark the deleted nodes for subsequent use.

After deleting the non-executable nodes, we get a reduced program (in the form of a reduced PDG). A simple data flow analysis of this reduced program determines the substitutable definitions. Really, all those definitions d_s^x for which there does not exist any du pair can be substituted by definition d_p^x, since these definitions do not affect any part of the program assuming that the principal definition is traversed.

The dynamic condition requires that a principal definition d_2^x should always be traversed at every possible iteration. It is not enough for d_2^x to be executed once after executing d_1^x. Namely, if at some iteration d_2^x is not traversed, but a du pair (d_1^x, u^x) is covered, u^x can influence the current predicate, even after d_2^x has been executed in a subsequent iteration.

Obviously, the static and dynamic conditions illustrated above are sufficient but not necessary.

To summarise, the procedure for determining all the substitutable definitions d_i^x for a given (principal) definition d_p^x (according to the static condition) is depicted in Figure 3.

```
procedure SubstitutableDefinitions
  input(PDG: PDG of static slice; q: current predicate;
        $d_p^x$:principal definition)
  output(Subst($d_p^x$) =  the set of substitutable defs for $d_p^x$ )
    Subst($d_p^x$) = $\emptyset$
    reduce $PDG$ eliminating non-executable nodes and arcs for $d_p^x$
    mark removed $PDG$ nodes for $d_p^x$
    derive reduced $G'(M)$       {$G'(M)$ is the ddgraph related to the
                                  reduced PDG}
    for each definition $d_i^x$ do
      if  there exists a du pair ($d_i^x$, $u^x$) for any $u^x$
          then SubstOK = false else SubstOK = true endif
      if SubstOk then insert $d_i^x$ into Subst($d_p^x$) endif
    endfor
endprocedure
```

Fig. 3. Procedure SubstitutableDefinitions

Using procedure `SubstitutableDefinitions` we may collect a set *Subst* (which may be empty) of substitutable definitions for each definition in a given module M. Those definitions yielding a not empty *Subst* set are the principal definitions. Principal definitions are contained in principal instructions. However, principal instructions may be located in different paths. The next step is therefore to group together the principal definitions whose instructions are located along the same path. For instance, for module *Example* in Fig. 1, procedure `SubstitutableDefinitions` finds two principal definitions. The first is (7) which may substitute (4) and (5), the second is (4) which substitutes (1). Since (4) and (7) may be in the same path we have only one list $Q =< (7), (4) >$.

More in general, we derive ordered sets, or *lists*, of principal instructions located along the same path in Postorder. To do this, we can use the results of the earlier principal definition analysis. We investigate principal definitions in Postorder. The first definition d_1 is inserted into a first list Q_1. Next, definition d_2 is investigated as follows. If d_2 reaches[3] d_1, then they are on the same path and d_2 is also inserted into Q_1; otherwise, d_2 is put into a new list. Any new element is inserted behind the last element of the list, the order of the elements within one class is thus Postorder. This method is repeated, i.e., for each principal definition d_i all the already formed lists are investigated. If d_i reaches the last element d_k in a class, then it is inserted into this list. If there is no suitable list

[3] Note that "reaches" here corresponds to graph reaching, rather than "reaching" definitions.

for the insertion of d_i, then a new list may be created in the following way. We investigate every element in each list (bottom-up) until we find a d_j such that d_i reaches d_j. If such a definition exists, then d_i and all the definitions in the list from the first to d_j are inserted into the new list. If no new list is formed in this way, then a new one with the unique element d_i is created.

Now we have a set of principal definitions collected into a number *num* of lists, whereby all the definitions in a list can be traversed by the same path. The final step of the principal slicing algorithm is the reduction of the *PDG*, based on the principal definition lists. Each list gives rise to a different principal slice, so we can derive a number *num* of principal slices.

We process the lists one by one. Let Q be the list considered. We can eliminate two types of nodes from the *PDG* (they correspond to the two cases in which a definition has no effect on the current predicates, as explained earlier). Informally, we can eliminate: Case 1) all the nodes corresponding to substitutable definitions for the principal definitions in Q; and Case 2) all the nodes corresponding to branches that are not executed because the principal instructions in Q are traversed (e.g., the *else* part of an *if* statement, where the *then* part corresponds to a principal definition).

PDG reduction according to Case 2) has already been considered when we discussed the removal of the non-executable nodes introducing procedure `SubstitutableDefinitions`. When there are several principal definitions in the list Q considered, all non-executable nodes for each principal definition are deleted. We now discuss Case 1) in more detail.

We consider the elements in Q from the first to the last (remember that this corresponds to a Postorder selection of principal instructions). For each principal definition in Q, we consider the associated set of substitutable definitions, or rather the set of nodes in *PDG* representing the substitutable definitions. All these nodes, with their incoming and outgoing arcs, are deleted from the *PDG*. After this, each node (with its incoming arcs) that has no outgoing arcs (but originally had some in the *PDG*) is repeatedly deleted from the *PDG* until there are no such nodes. Whenever a substitutable definition is also a principal definition, then the substitutable definition is removed from the analyzed class. If a list is modified, then it is compared to the as yet not processed lists and equivalent lists (if any) are removed.

As a result of the above reduction, we get a principal slice *PS* for the current predicate in the form of a reduced *PDG*. From the reduced *PDG* and the original module, the program segment corresponding to the principal slice can be derived. Note that since the principal slicing method involves subalgorithms (for example, du pair determination) that are safe but not precise, we only obtain a slice with a near minimum number of influencing predicates. For this slice, we can then determine a path from entry to the current predicate (e.g., one containing a near minimum number of predicates, using static methods based on control flow analysis). Let us refer to this path as p_{min}. We may construct p_{min} for each list Q_i and obtain a set of paths (for instance we may consider them ordered by increasing sizes). Very plausibly, some of them are feasible.

The listing of the described algorithm for generating the principal slices for the current predicate q is depicted in Fig. 4.

```
program PrincipalSlices
  input(M: program module; q: current predicate)
  output($\mathcal{PS}$: set of pairs [$PDG_i$ of a principal slice, list $Q_i$
                                    of principal instructions] )
  begin
  derive $G(M)$
  compute  static slice S for M
  derive $PDG$ (with labels)
  $\mathcal{PS}$ = $\emptyset$
  for each definition $d_j^x$ in $PDG$ do  {$d_j^x$ is a possible principal
                                         definition of $x$}
    call SubstitutableDefinitions(M,q,$d_j^x$)  { it derives Subst($d_j^x$)}
  endfor
  group principal definitions $d_j^x$ into lists $Q_1$,..., $Q_{num}$
  for i = 1 to $num$ do
    select list $Q_i$ of principal definitions
    reduce $PDG$ to $PDG_i$ by eliminating:
      all the marked non-executable nodes,
      all the substitutable nodes for each $d_j^x \in Q_i$ and then transitively
      all nodes that have no leaving arcs
   $\mathcal{PS}$ = $\mathcal{PS}$ $\cup$ {[$PDG_i$,$Q_i$]}
  endfor
endprogram
```

Fig. 4. Program PrincipalSlices

Algorithm PrincipalSlices is now illustrated by applying it to Fig. 1 for criterion C_p=($<(7),(4)>$, (e), {x}) (the list $Q=<(7),(4)>$ has been previously derived by procedure SubstitutableDefinitions). First we process (7). *PDG* reduction for (7) removes (9) as a non-executable node and (4), (5) as substitutable nodes. Thus as yet not processed principal definition (4) is removed from Q. Deleting (9) involves the removal of (6) and (transitively) of (c). Ignoring (4) and (5) (in addition to (6) and (c)), (b) can also be removed. Removing (c), (6) and (9) involves the deletion of (2) and (3) and then transitively of (a). Finally, since (5) is deleted, (1) can also be removed obtaining a principal slice only consisting of lines (0),(d), (7) and (e). Thus, for this example, using principal slicing we can see that only one predicate actually needs to be satisfied to generate a test path reaching instruction (10), precisely predicate (d)=$z+y>10$. By contrast, existing static approaches [3, 21], which only use control flow analysis, could only select a "short" control path by taking the *then* branch at predicate (b) (i.e., only avoiding predicate (c)).

The *PDG*s for the original and the sliced programs are shown in Fig. 5.a and 5.b.

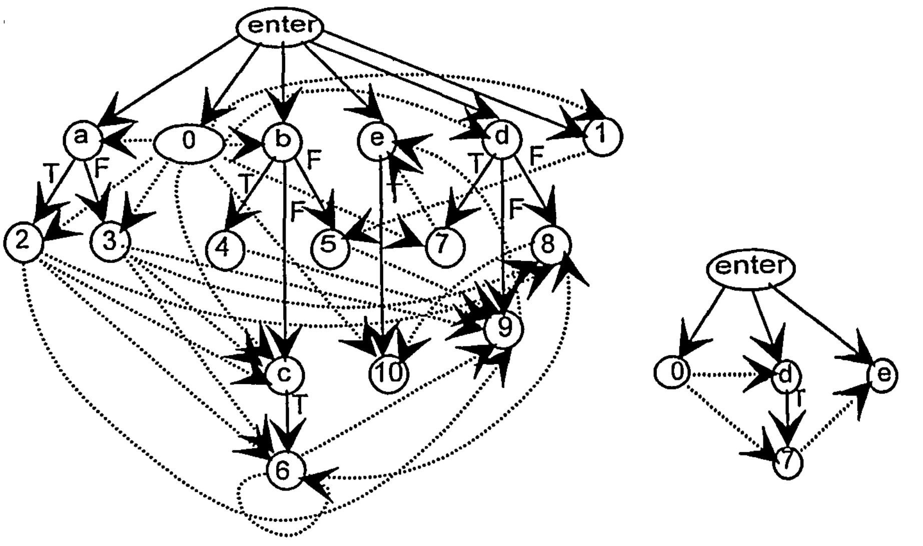

Fig. 5. PDG *a* for procedure Example and *b* for its principal slice

We omit here the complexity analysis of algorithm `PrincipalSlices` because some steps of the algorithm can be realized in different ways resulting in different precision. On the other hand, we observe that our algorithm leads to an exponential improvement of feasible path determination, therefore any polynomial algorithm is efficient, and any step of `PrincipalSlices` can be realized by polynomial subalgorithms.

4 FEASIBLE TEST PATH SELECTION FOR BRANCH COVERAGE

In this section we illustrate the application of principal slicing to white–box testing. To do this we choose as an example a particular criterion, specifically branch coverage, for two reasons. Firstly, branch coverage is certainly the best suited white–box criterion for exemplificative purposes, because it is well known and very intuitive. It requires that each branch alternative in the program control flow is executed at least once. Secondly, branch testing is widely used in industry, and is accepted as the "minimum mandatory testing requirement" for the unit test phase by major software companies [2]. Hence, the proposed example procedure may be of interest to software test practitioners who wish to improve branch test automation.

The procedure described is not only an example of using principal slicing, but it also provides an optimized method for the automatic generation of a test suite satisfying the branch coverage criterion. The latter statement is justified by the fact that we use the concept of *spanning sets* of entities for coverage criteria, recently introduced in [14].

Coverage test criteria require that a set E_c of entities of the program (control or data) flow is covered by the executed tests. A spanning set UE is a subset of E_c with the following properties:
i) a test suite which covers all the entities in this subset UE will cover all the entities E_c in the program (control or data) flow, and
ii) it is a minimum subset of E_c with property i), i.e., any subset of E_c having property i) above has cardinality at least $|UE|$.

In particular, for the branch coverage criterion the entities to be covered are represented by arcs in the program ddgraph, and there exists only one spanning set of arcs for a given ddgraph. A test suite covering all the arcs in the spanning set will also cover all the arcs in the ddgraph, and no smaller subset of arcs exists with this property. Two different methods for deriving a spanning set of arcs for the branch coverage criterion are given in [3] and in [14], respectively.

In Figure 6 we present the procedure FIND-A-TEST-SUITE-FOR-BRANCH-COVERAGE (FTSBC) which finds a set of test inputs to cover a set of arcs CA in a given program ddgraph G. The set CA can correspond either to the whole set of program branches, or to the set of branches as yet not covered by the already executed tests. Below, we discuss some steps of this procedure needing further explanation.

```
procedure FIND-A-TEST-SUITE-FOR-BRANCH-COVERAGE
input ($G$: program ddgraph; $CA$: set of ddgraph arcs to be covered)
output ($T$: set of test data covering $CA$)
begin
 $T = \emptyset$; {$T$, initially empty, will return the test suite found}
 1. derive the spanning set $UE$ of arcs for program ddgraph $G$;
 2. $CUA = CA \cap UE$;
 3. select an arc $U_i \in CUA$;
 4. derive a set of principal slices $\mathcal{PS}_i$ for predicate $T(U_i)$;
 5. repeat
      (a) select a slice $S_j^i$ in $\mathcal{PS}_i$;
      (b) derive a path $p_j^i$ over $S_j^i$ and look for an input $t_j^i$ which executes $p_j^i$
    until a $t_j^i$ has been found or there are no more slices to be processed;
 6. $T = T \cup \{t_j^i\}$; {assuming a $t_j^i$ was found}
 7. execute the program on $t_j^i$ and monitor the set of arcs $A_i$ exercised by
    $t_j^i$;
 8. $CUA = CUA - CUA \cap A_i$;
 9. if $CUA = \emptyset$ then return($T$) else goto 3;
endprocedure.
```

Fig. 6. Procedure FTSBC

Step 3: Procedure FTSBC finds test cases one at a time. Once the spanning set of arcs has been found, we need to choose an as yet not covered arc in it to be processed next. We would like to use a selection heuristic which reduces the effort of generating test inputs in the further steps of FTSBC. To this end, we observe the following points: i) the required time of steps 4 and 5 of procedure FTSBC increases with the number of predicates between the entry arc and the selected arc U_i; and ii) a test path covering the selected arc U_i may in general cover more arcs in the spanning sets after U_i. Therefore, we choose to select an arc U_i from the set of as yet not covered arcs CUA so that U_i is "close" to the program entry (i.e., the subpath from the entry to this arc has a low number of predicates). A detailed procedure can be found in [3].

Step 4: Once a U_i has been selected, we can use the program *PrincipalSlices* from the previous section with $T(U_i)$ as the current predicate for deriving a set of principal slices.

Step 5: Hence, we select a principal slice, for instance the slice S^i_{small} that has the smallest number of predicates. However, there may be several paths over this slice. We select a path p^i_{short} from these paths such that the number of predicates along it is near minimum. To do this we can apply one of the methods in the literature, e.g., [3], [21]. We can also apply one of the methods in the literature (either symbolic execution [4] [9] or execution oriented approach [11]) to find an input that executes such a path. Considering that path p^i_{short} has a near minimum number of influencing predicates (because of both principal slicing and path minimization process), these methods have high chances of success.

The remaining steps of the procedure are straighforward.

A practical question arises with loops. Usually, the shortest paths over principal slices iterate loops zero times. However in some cases, though zero iteration of some loops is feasible, this doesn't happen if several loops are present and have to be combined. In these cases, our method would lead to infeasible paths, at least at the beginning of the analysis. To avoid this, we can modify both the ddgraph and the *PDG* of the module to be tested such that we force the execution of at least one iteration for the loops (except possibly for the current predicate). Considering structured programs this transformation can easily be done. The resulting principal slice can be then converted back based on the original module.

In the example provided, principal slicing is used to derive a path to reach a certain point. We observe that the `PrincipalSlices` algorithm can easily be adjusted to find paths which traverse more than one given point, as would be required for instance to cover a given du pair in data flow testing. In this case, we find a principal slice to the last (i.e., the nearest to the exit arc) point and initialise each list Q_i with the set of the other points to be reached (this would require also to modify the insertion procedure in order to preserve Postorder). In fact, this is sufficient to guarantee that all of them are traversed by any path in the derived principal slices.

5 CONCLUSIONS

In this paper a new method for selecting feasible program paths has been introduced. Previous methods either considered only control flow information, such as the static selection of shortest paths [21], [3] and the goal-oriented approach [12], or used data flow without any optimization [7]. As statistically validated [20], feasible paths can be found with a much higher probability along paths traversing fewer predicates. By applying our new method of principal slicing, paths with almost minimum number of influencing predicates can be found such that both control and data flow are considered.

Principal slicing is a general method for test data generation since it finds the best potentially feasible paths in any program. This is the basis for every automatic path–wise test data generation tool, independently of which testing criterion is applied. Thus, by applying principal slicing, automated branch, data flow, and even constraint-based testing can be improved.

Another advantage of principal slicing is that all path realisation methods can use it. This means that after the selection of a very likely feasible path, both symbolic execution and function minimization methods can be applied.

We have also provided an optimised procedure for automating branch coverage testing, using principal slicing and the concept of a spanning set of branches. The procedure derives a minimal test suite (since we use the spanning set of branches to guide test path selection, we avoid redundant tests) in an effective way (since the paths are derived over principal slices with a near minimum number of influencing predicates, the test paths found are very likely to be feasible).

An interesting observation is that the concept of principal slicing provides an appealing generalization of program slicing, in the following sense. Static slices, involving *any statement* that affects a given predicate at a certain instruction, and dynamic slices, involving influencing statements along *just one* path (pertaining to a specified input), can be seen as two extreme approaches to perform program slicing. In between, one could imagine several other slicing methods, yielding slices which involve a set of paths which is smaller than for static slices and larger than for dynamic slices. These methods could be based on specific criteria depending on the objective the slice is being constructed for. In other words, one could tailor and optimise program reduction to one's specific needs, in a sort of *special–purpose* slicing. There have been some proposals in the literature. Venkatesh [16] has introduced *quasi–static* slicing, in which values of some of the inputs are fixed, while values of other are arbitrary. This can be used for program understanding. Agrawal et. al. [1] introduced *relevant slices* for incremental regression testing. Relevant slices are dynamic slices determined as follows: (1) select an input and execute the program; (2) assume that any statement in the program may be modified. A relevant slice contains all the influencing statements that may be traversed if (1) and (2) hold.

In our case, principal slicing derives a special-purpose slice which is optimised to facilitate *feasible* path generation for path-wise testing. We have discussed how, with regard to this objective, principal slices are more suitable than both static and dynamic slices and how their use in test input generation improves on earlier approaches.

References

1. H. Agrawal, J.R. Horgan, E.W. Krauser and S.A. London. Incremental regression testing, *Proc. of the 1993 IEEE Conf. on Software Maintenance*, Montreal, Canada, 348-357 (1993)
2. B. Beizer. *Software Testing Techniques, Second Edition.* Van Nostrand Reinhold, New York. 1990.
3. A. Bertolino and M. Marré. Automatic generation of path covers based on the control flow analysis of computer programs. *IEEE Trans. on Software Eng.*, 20(12):885–899, (1994).
4. L. A. Clarke. A system to generate test data and simbolically execute programs. *IEEE Trans. on Software Eng.*, 2(3):215-222, (1976).
5. R. Conradi, Experience with FORTRAN VERIFIER - A tool for documentation an error diagnosis of FORTRAN-77 programs *Proc. 1st European Software Engineering Conference*, Strasburg, France 8-11 Sept. 263-275 Springer Verlag LNCS 289 (1987)
6. R. A. DeMillo and A.J. Offutt. Constraint–based automatic test generation. *IEEE Trans. on Software Eng.*, 17(9):900–910, (1991).
7. R. Ferguson and B. Korel. The chaining approach for software test data generation. *ACM Trans. on Software Eng. and Meth.*, 5(1):63-86, (1996)
8. S. Horwitz, T. Reps and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Progr. Lang. Syst.*, 12(1):26–61, (1990).
9. W. E. Howden. Symbolic testing and the DISSECT symbolic evaluation system. *IEEE Trans. on Software Eng.*, 3(4):266–278, (1977).
10. M. Kamkar. An overview and comparative classification of program slicing techniques. *Journal of Systems and Software*, 31(3):197-214, (1995)
11. B. Korel. Automated software test data generation. *IEEE Trans. on Software Eng.*, 16(8):870–879, (1990).
12. B. Korel. Dynamic method for software test data generation. *J. Softw. Testing Verif. Reliab.* 2(4):203-213, (1992)
13. B. Korel and J. Laski. Dynamic slicing of computer programs. *Journal of Systems and Software*, 13(3):187–195, (1990)
14. M. Marré and A. Bertolino. Reducing and estimating the cost of test coverage criteria. In *Proc. ACM/IEEE Int. Conf. Software Eng. ICSE-18*, pages 486–494, Berlin, Germany, March 1996.
15. K.J. Ottenstein and L.M. Ottenstein. The program dependence graph in a software development environment. *ACM SIGPLAN Notices*, 19(5):177-184, (1984)
16. G.A. Venkatesh. The semantic approach to program slicing *Proc. of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation* Toronto, Canada, 107-119 (1991)
17. M. Weiser. Program slicing. *IEEE Trans. on Software Eng.*, 10(4):352–357, (1984).
18. E. Weyuker. Translatability and decidability questions for restricted classes of program schemas. *SIAM Journal on Computers*, 8(4):587–598, (1979)
19. L. J. White and E. I. Cohen. A domain strategy for computer program testing. *IEEE Trans. on Software Eng.*, 6(3):247–257, (1980).
20. D. F. Yates and N. Malevris. Reducing the effects of infeasible paths in branch testing. *ACM SIGSOFT Software Engineering Notes*, 14(8):48–54, (1989).
21. D. F. Yates and N. Malevris. The effort required by LCSAJ testing: an assessment via a new path generation strategy. *Software Quality J.*, 4(3):227–242, (1995).