Information Processing Letters 29 (1988) 155–163
North-Holland

26 October 1988

DYNAMIC PROGRAM SLICING *

Bogdan KOREL

Department of Computer Science, Wayne State University, Detroit, MI 48202, U.S.A.

Janusz LASKI

Department of Computer Science and Engineering, Oakland University, Rochester, MI 48063, U.S.A.

Communicated by W.L. Van der Poel
Received 11 September 1987

A dynamic program slice is an executable subset of the original program that produces the same computations on a subset of selected variables and inputs. It differs from the static slice (Weiser, 1982, 1984) in that it is entirely defined on the basis of a computation. The two main advantages are the following: Arrays and dynamic data structures can be handled more precisely and the size of slice can be significantly reduced, leading to a finer localization of the fault. The approach is being investigated as a possible extension of the debugging capabilities of STAD, a recently developed System for Testing and Debugging (Korel and Laski, 1987; Laski, 1987).

Keywords: Slicing, dynamic slice, trajectory, data dependence, control dependence, debugging

1. Introduction

A slice S of a program P is an executable subset of P that computes the same function as P does in a subset of variables, at some selected point of interest [19,23,24]. Slicing has been shown useful in program debugging by narrowing the size of the suspected piece of incorrect code. As originally introduced [23,24], it is a static concept: it involves all potential terminating program executions, including those which are infeasible. In debugging practice, however, we typically deal with a particular incorrect execution and, consequently, are interested in locating the cause of incorrectness (programming fault) of *that* execution. For this reason we are interested in a slice that preserves the program's behavior for a specific input, rather than that for the set of all inputs for which the program terminates. This type of program slice, which we call a *dynamic* one, is introduced in this paper.

It is shown that dynamic slicing provides a finer localization information. A static slice very often contains statements which have no influence on the values of variables of interest. A dynamic slice can be considered a refinement of the static one: By applying dynamic analysis it is easier to identify those statements in the static slice which do not have influence on the variables of interest. By reducing the searching space for the fault in the program, one can more efficiently localize it.

In this paper we also investigate the dynamic handling of arrays in slicing. In the static approach, an entire array is treated as a single variable, i.e., each definition or use of any array element is treated as a definition or use of the entire array. While this method is easy to implement, it fails to take into account any information about particular array elements. This can lead to the inclusion of statements which do not have any influence on the values of certain array elements.

* This research was partly supported by the National Science Foundation under Grant No. ECS-82-18072.

As a result, the slice can be unnecessarily large. In our approach, every array element is treated as a separate variable. This is due to the fact that, during program execution, it is possible to determine the value of an array subscript and, therefore, to determine which array elements are used or modified at every point of program execution. In the concluding Section 5 we also comment on a possible application of the method to other structured data.

The reader is assumed to be familiar with the original static concept of program slicing [23,24]. In what follows, + stands for set union.

2. Background

A *flowgraph* of a program P is a directed graph $C=(N, A, en, ex)$, where N is a set of *nodes*, A is a binary relation on N (a subset of $N \times N$) referred to as the set of *arcs*, and *en* and *ex* are, respectively, a unique entry and a unique exit node, $en, ex \in N$.

For the sake of simplicity we restrict our analysis to a subset of structured PASCAL-like programming language constructs, namely: sequencing, **if–then–else**, and **while**-loop. A node in N corresponds to a smallest, not further decomposable, single-entry single-exit executable part of a statement in P, referred to as an *instruction*. It can be, for example, an assignment statement, an input or an output statement, or the ⟨expression⟩ part of an **if–then–else** or **while** statement, in which case it is called a test instruction.

An arc $(n, m) \in A$ corresponds to a possible transfer of control from instruction n to instruction m. A *path* from the entry node *en* to some node l, $l \in N$, is a *finite* sequence $\langle n_1, n_2, \ldots, n_q \rangle$ of instructions, such that $n_1 = en$, $n_q = l$, and (n_i, n_{i+1}) is in A for all n_i, $1 \leqslant i < q$. If $n_q = ex$, then the path is a *program path*. A path is *feasible* if there exists input data which causes the path to

```
var n, i, j, p: integer; a: array[1..10] of integer;
 1      input(n,a);
 2      i := 1;
 3      while i < n do begin
 4            min := a[i];
 5            p := i;
 6            j := i + 1;
 7            while j < = n do begin
 8                  if a[j] < min then begin
 9                     min := a[j];
10                     p := j;
                       end;
11                  j := j + 1;
                    end;
12            a[p] := a[i];
13            a[i] := min;
14            i := i + 1;
      end;
15    output(a);
```

Fig. 1. A sorting program.

Instruction number	Instruction text	
1^1	input(n, a)	
2^2	$i := 1$	
3^3	$i < n$	
4^4	min $:= a[i]$	/ * min $:= a[1]$ * /
5^5	$p := i$	
6^6	$j := i + 1$	
7^7	$j <= n$	
8^8	$a[j] <$ min	/ * $a[2] <$ min * /
11^9	$j := j + 1$	
7^{10}	$j <= n$	
12^{11}	$a[p] := a[i]$	/ * $a[1] := a[1]$ * /
13^{12}	$a[i] :=$ min	/ * $a[1] :=$ min * /
14^{13}	$i := i + 1$	
3^{14}	$i < n$	
15^{15}	output(a)	

Trajectory $T = \langle 1, 2, 3, 4, 5, 6, 7, 8, 11, 7, 12, 13, 14, 3, 15 \rangle$

Fig. 2. A trajectory of the program from Fig. 1 on input data $n = 2$, $a = (2, 4)$.

be traversed during program execution. A feasible path that has actually been executed for some input will be referred to as a *trajectory*. For example, if the program in Fig. 1 is executed on the input $i = (n, a) = (2, (2, 4))$, the trajectory T in Fig. 2 is traversed. An executed program path is a *program trajectory*. Observe that a trajectory can be an initial (finite) segment of an 'infinite' path if the execution involved does not terminate.

In the deterministic case, the trajectory is uniquely determined by the input while in the nondeterministic case (e.g., referencing an uninitialized variable) there might be many trajectories for the same input. In either case, however, there are many inputs that give rise to the same trajectory. In what follows we assume the deterministic case.

Notationally, T is an abstract list [9] whose elements are accessed by position, for example, for T in Fig. 2 we have $T(5) = 5$, $T(9) = 11$, and $T(14) = 3$. To handle multiple occurrences of the same instructions in the trajectory (for instance, instruction 3 appears twice in T in Fig. 2), every instruction is characterized by its position in the sequence. Let $N(T)$ be the set of pairs (instruction in T, its position in T) defined as follows:

$$N(T) = \{(X, p) : X \in N, T(p) = X\}.$$

An (X, p) will be written down as X^p and interpreted as "an instruction X at the execution position p". For instance, 3^3 and 3^{14} are two occurrences of instruction 3 in the trajectory T shown in Fig. 2.

The following dataflow concepts are of dynamic nature because they are defined with respect to the trajectory T, rather than to the flowgraph itself.

A *use* of variable v is an instruction X^p in which this variable is referenced. A *definition* of variable v is an instruction X^p which assigns a value to that variable.

In the framework of static program analysis, an assignment to an array element is treated as a definition of the entire array. This does not seem a real obstacle in program optimization, the first area of application of data flow analysis [1,2,3,7, 10]. It does cause serious problems, however, in

```
1  i := 1;
2  j := 2;
3  read(k);
4  while i < 3 do begin
5     a[i] := a[j] * a[k];
6     i := i + 1;
7     j := j + 2;
8     k := k + 3;
   end;
```

$T = \langle 1, 2, 3, 4, 5, 6, 7, 8, 4, 5, 6, 7, 8, 4 \rangle$

Fig. 3. A sample program and a trajectory for $k = 2$.

data flow testing and debugging, where it is highly desirable to identify the particular array entries manipulated by the program [11–17]. It is precisely what the dynamic approach makes possible. In it, every array element is treated as a separate variable. For example, the following is the set of all variables in the program in Fig. 1,

$$\{n, i, j, p, \min, a[1], a[2], \ldots, a[10]\}.$$

Let $U(X^p)$ be the set of variables whose values are used in X^p and $D(X^p)$ be the set of variables whose values are defined in X^p. It is essential to observe that, unlike their static counterparts, these sets are dynamic. Clearly, given two occurrences X^p and X^q of the same instruction X, these sets might be different in each case. If X handles an array, and every array element is treated as a separate variable, then at each execution of X different entries in the array might be used or defined. For example, during the first execution of instruction 5 of the program of Fig. 3 the following variables are used and defined:

$$U(5^5) = \{i, j, k, a[2]\}, \qquad D(5^5) = \{a[1]\}.$$

However, during the second execution of the same instruction, we have

$$U(5^{10}) = \{i, j, k, a[4], a[5]\},$$
$$D(5^{10}) = \{a[2]\}.$$

The dynamic nature of the sets U and D is in contrast with their counterparts in static analysis. In the case of our simple language it is due to the dynamic treatment of arrays. For example, the static analysis of the program in Fig. 3 would render

$$U(5) = \{a, i, j, k\}, \qquad D(5) = \{a\}.$$

It is worth noting, however, that if nodes in the graph correspond to procedure calls or some single-entry single-exit compound statements rather than instructions, the dynamic parts of the sets U and D might also involve scalar variables.

It is assumed that the sets U and D for each instruction in the trajectory can be identified through program instrumentation.

2.1. Definition. Given a trajectory T, an instruction X^p is the *last definition* of variable v at execution position q in T iff $v \in D(X^p)$ and, for all k, $p < k < q$, $v \notin D(Y^k)$.

The last definition X^p of v at q is then the unique instruction which has last assigned a value to variable v when q is reached on T. For instance, in the execution trace of Fig. 2, 2^2 is the last definition of variable i at execution position 6.

The last definition is also a *reaching* one in the static sense [8]. The opposite, however, is not necessarily true. A reaching definition is defined in terms of the flowgraph only, rather than in those of a trajectory. Therefore, it is only potentially last and, moreover, it might also be infeasible.

3. Dynamic slice

Intuitively, a dynamic slice is an executable part of the program whose behavior is identical to that of the original program with respect to a subset of variables of interest and at execution position q. Such a 'view of interest' is captured by the following definition.

3.1. Definition. Let T be the trajectory of program P on input x. A *slicing criterion* of program P

executed on input x is a triple $C=(x, I^q, V)$, where I is an instruction at position q on T and V is a subset of variables in P.

Observe that the corresponding static slicing criterion [24] is just a pair (I, V). Clearly, our slicing criterion is defined w.r.t. a given trajectory on a specific input x rather than w.r.t. the set of all possible paths in the flowgraph. Yet another difference is in the interpretation of the 'position of interest'. In the static case, this is instruction I in P; in our case, this is instruction I at execution position q in trajectory T.

Typically, the set V contains those variables in the program that have been found incorrect at q. The slice is then used to locate the cause of the incorrectness. V might be, however, a set of variables that are correct at q, too; the slice might then be used for verification purposes.

To formally define the dynamic slice we need some definitions of list operations.

Let $T=\langle X_1, X_2, \ldots, X_m\rangle$ be a trajectory of length m and let q be a position in T, $1 \leqslant q \leqslant m$. By $F(T, q)$ we denote the *front* of T w.r.t. q, i.e., a sublist $\langle X_1, X_2, \ldots, X_q\rangle$, containing the first q elements of T [9]. Correspondingly, $B(T, q)$, the *back* of T w.r.t. q, is the sublist $\langle X_{q+1}, \ldots, X_m\rangle$, containing elements that follow $T(q)$. We have, of course, $T=F(T, q) \,||\, B(T, q)$, where $||$ stands for the concatenation of lists. By $DEL(T, r)$, where r is a predicate on the set of instructions in T, we mean a *subtrajectory* obtained from T by deleting from it all elements $T(i)$ that satisfy r. In other words, $DEL(T, r)$ is the result of an exhaustive application of the delete operation to elements $T(i)$ that satisfy $r(T(i))$.

3.2. Definition. Let $C=(x, I^q, V)$ be a slicing criterion of program P and T a trajectory of P on input x. A *dynamic slice* of P on C is any executable program P' that is obtained from P by deleting zero or more statements from it and, when executed on input x, produces a trajectory T' for which there exists an execution position q' such that:

(1) $F(T', q')=DEL(F(T, q), T(i) \notin N'$ and $1 \leqslant i \leqslant q)$,

(2) for all $v \in V$, the value of v before the execution of instruction $T(q)$ in T equals the value of v before the execution of instruction $T'(q')$ in T',

(3) $T'(q')=T(q)=I$,

where N' is a set of instructions in P'.

The following observations clarify some salient properties of the dynamic slice.

First, a dynamic slice partially replicates the front of T (w.r.t. q). Clearly, we are interested in the partial reproduction of the behavior of program P up to the execution position q. Moreover, dynamic slice preserves the number of occurrences of instructions in the trajectories T and T'. For instance, if a loop in P iterates five times, then we require that the same loop, if included in P', also iterates five times. But, these requirements do not necessarily hold for the back of T and T' (past the execution positions q and q', respectively). Indeed, the absence of some instructions in P' might cause unpredictable control flow patterns in T' when the execution continues past q'.

Second, it is required that instruction I^q appear in the slice. This is in contrast to the static definition of slice in which that instruction does not necessarily appear [24]. Our experience with static slices shows that the programmer can be lost if statement I is not included in the slice, particularly if I is in a loop. We feel therefore that including it into the slice is more realistic for debugging purposes.

Third, the fact that all variables in V have the same values at q in T and at q' in T' does not necessarily guarantee that variables not in V will have the same values at those positions nor that those in V itself will have the same values along T and T' (except at q and q').

There can be many different dynamic slices for a given program and a slicing criterion, and there is always at least one such a slice: The entire program itself.

3.3. Definition. Let C be a slicing criterion of program P executed on input x. A dynamic slice DS of P on C is *statement-minimal* if no other dynamic slice of P on C has fewer statements than DS.

As in the case of static slices, the problem of finding statement-minimal dynamic slice is undecidable [24]. However, data flow analysis can be used to construct conservative slices, guaranteed to have the slice properties but with, perhaps, too many statements.

4. Finding dynamic slice

Intuitively, given a slicing criterion $C = (x, I^q, V)$, a dynamic slice contains only those instructions from N that (i) influence the variables in V at q, and (ii) appear in T. Data flow analysis along T can help in finding them by tracing backwards some well-defined dependencies between instructions in T. This can be done in two steps. First, find a subtrajectory T' of T that meets the criteria of Definition 3.2 and then reconstruct a P' from T'.

The identification of T' is equivalent to finding a subset of $N(T)$ that contains all instructions in the trajectory T which have influence on V at q and guarantee that I^q is reached in the first place. Such a subset will be referred to as the *slicing set*, denoted S_C.

To capture the intuitive notion of influence we introduce two types of dependence relations between program instructions in the trajectory T. These relations formally define the properties that instructions in $N(T)$ must meet to be in a slice. The relations are constructive in the sense that they can be used to formulate an algorithm, however inefficient it might be [11].

Let $C = (x, I^q, V)$ be a slicing criterion. In what follows we introduce two types of influences (dependences) between instructions in the front of T w.r.t. q. Those are the Data–Data and Test–Control binary relations on $N_C(T)$, where

$$N_C(T) = \{ X^p : X^p \in N(T) \text{ and } 1 \leqslant p \leqslant q \}.$$

Clearly, we are only interested in the instructions in the front of the trajectory T, up to the execution position q.

The DD (Data–Data) Relation. The DD relation models a situation where one instruction assigns a value to an item of data and the other instruction uses that value. For instance, in the execution trace of Fig. 2, instruction 2^2 assigns a value to variable i and instruction 6^6 uses that value.

$\mathrm{DD}(1^1) = \{3^3, 4^4, 7^7, 8^8, 7^{10}, 12^{11}, 3^{14}, 15^{15}\}$
$\mathrm{DD}(2^2) = \{3^3, 4^4, 5^5, 6^6, 12^{11}, 13^{12}, 14^{13}\}$
$\mathrm{DD}(4^4) = \{8^8, 13^{12}\}$
$\mathrm{DD}(5^5) = \{12^{11}\}$
$\mathrm{DD}(6^6) = \{7^7, 8^8, 11^9\}$
$\mathrm{DD}(11^9) = \{7^{10}\}$
$\mathrm{DD}(13^{12}) = \{15^{15}\}$
$\mathrm{DD}(14^{13}) = \{3^{14}\}$

Fig. 4. The DD relation for the trajectory of Fig. 2 and the slicing criterion $C = (x, 15^{15}, \{a[2]\})$, where $x = (n, a) = (2, (2, 4))$. *Notation*: $\mathrm{DD}(k) = \{l : k \ \mathrm{DD}\ l\}$.

DD is a binary relation on $N_C(T)$ defined as follows:

> X^p DD y^t, $1 \leqslant p < t \leqslant q$, iff there exists a variable v such that: (1) $v \in U(y^t)$, and (2) X^p is the last definition of v at t.

Fig. 4 shows the DD relation for the trajectory in Fig. 2 and the slicing criterion $C = (x, 15^{15}, \{a[2]\})$.

Observe that, appearances to the contrary, the DD relation is *not* a subset of the set of static definition-use chains [7].

The TC (Test–Control) Relation. The TC relation captures the dependence between test instructions and the instructions which can be chosen to execute or not execute by these test instructions. For instance, test instruction 8 in the program of Fig. 1 has 'influence' on the execution of instruction 9, but it has no influence on the execution of instruction 11. To define the TC relation, we need the following notion of the *scope of influence* for the **if** and **while** statements [11].

(a) **if** X **then** $B1$ **else** $B2$; instruction Y is in the scope of influence of X iff Y appears in $B1$ or $B2$,

$$TC(3^3) = \{4^4, 5^5, 6^6, 7^7, 8^8, 11^9, 7^{10}, 12^{11}, 13^{12}, 14^{13}, 3^{14}\}$$
$$TC(7^7) = \{8^8, 11^9, 7^{10}\}$$
$$TC(3^{14}) = TC(7^{10}) = \{\ \}$$

Fig. 5. TC relation for the program of Fig. 1 and the slicing criterion $C = (x, 15^{15}, \{a[2]\})$, $x = (n, a) = (2, (2, 4))$.

(b) **while** X **do** B; instruction Y is in the scope of influence of X iff Y is in B or $X = Y$.

In the program of Fig. 1, instruction 6 is in the scope of influence of test instruction 3, but instruction 15 is not in the scope of influence of instruction 3. Observe that the test instruction X of a **while**-loop is in the scope of influence of itself because every execution of X has influence on the next execution of X. For instance, in the trajectory of Fig. 2 the outcome of the test instruction 3^3 influences the execution of 3^{14}.

TC is a binary relation on $N_C(T)$ defined as follows:

X^p TC y^t, $1 \leqslant p < t \leqslant q$, iff (1) Y is in the scope of influence of X, and (2) for all k, $p < k < t$, $T(k)$ is in the scope of influence of X.

The TC relation for the trajectory in Fig. 2 is shown in Fig. 5.

The relations DD and TC capture the influences that exist between instructions in the trajectory. They fail, however, to guarantee that the number of occurrences of an instruction in T (between 1 and q) and T' (between 1 and q') are the same, a property that follows from condition (1) of Definition 3.2. Towards that goal we define the *Identity Relation* IR on $N_C(T)$ as follows:

X^p IR y^t, $1 \leqslant p,\ t \leqslant q$, iff $X = Y$.

For example, for the trajectory of Fig. 2 we have 3^3 IR 3^{14} and 7^7 IR 7^{10}; observe that IR is symmetric, for example 3^{14} IR 3^3 holds, too.

To find S_C we first find a set A^0 of all instructions that have a direct influence on V at q and on the execution of instruction I^q. We have

$$A^0 = LD(q, V) + LT(I^q),$$

where $LD(q, V)$ is the set of last definitions of variables in V at execution position q, and $LT(I^q)$ is the set of test instructions which have control influence on the execution of I^q. More formally, we can state

$$LD(q, V) = \{\, X^p \in N_C(T) : \text{there exists a } v \in V \text{ such that } x^p \text{ is the last definition of } v \text{ at } q \,\}$$

and

$$LT(I^q) = \{\, X^p \in N_C(T) : X^p \text{ TC } I^q \,\}.$$

We will find S_C iteratively, as a limit of the sequence $S^0, S^1, \ldots, S^n$, $0 \leqslant n < q$, defined as follows:

$$S^0 = A^0,$$
$$S^{i+1} = S^i + A^{i+1},$$

where

$$A^{i+1} = \{\, X^p \in N_C(T) :$$
(1) $X^p \notin S^i$, and
(2) there exists a $Y^t \in A^i$, $p, t \leqslant q$, X^p (DD + TC + IR) y^t $\}$.

The sets S^i, $i = 0, 1, \ldots, k$, can be thought of as an increasing sequence of successive approximations of S_C. Each S^i is bounded from above by $N_C(T)$. Eventually, because T is finite, there is an $A^{k+1} = \{\ \}$, for some k. If the above recursive definition is the basis for a corresponding search

process, that process will always terminate. According to the postulated properties of a dynamic slice in Section 3, instruction I^q must also be included in S_C. The following will guarantee its inclusion in the slice:

$$S_C = S^k + \{I^q\},$$

where S^k is the limit of sequence $\{S^i\}$.

The (disjoint) sets A^i, $i = 0, 1, \ldots, n$, contain those instructions that have i-level influence on V at q. Clearly, instructions in A^0 have direct influence on V at q. Instructions in A^i, $i > 0$, have indirect influence on V at q by directly influencing those in A^{i-1}. Intuitively, the slicing set contains instructions that have direct or indirect influence on V at q and the execution of I^q.

Given S_C, the slice is found in a straightforward way: Inspect instructions in P and select only those which appear at least once in the slicing set. Needless to say, to ensure syntactical correctness all necessary declarations are to be selected, too.

Example. Consider again the trajectory T in Fig. 2. For the criterion

$$C1 = (x, 15^{15}, \{a[2]\}),$$
$$x = (n, a) = (2, (2, 4))$$

we have

$$LD(15, \{a[2]\}) = \{1^1\}, \qquad LT(15^{15}) = \{\ \},$$
$$A^0 = \{1^1\}, \qquad S^0 = \{1^1\}, \qquad A^1 = \{\ \},$$
$$S_{C1} = S^0 + \{15^{15}\} = \{1^1, 15^{15}\},$$

and, finally, the dynamic slice

```
 1 input(n, a);
15 output(a);
```

For the slicing criterion

$$C2 = (x, 15^{15}, \{a[1]\})$$

we have

$$LD(15, \{a[1]\}) = \{13^{12}\}, \qquad LT(15^{15}) = \{\ \},$$
$$A^0 = \{13^{12}\}, \qquad S^0 = \{13^{12}\},$$
$$A^1 = \{2^2, 3^3, 4^4\},$$
$$S^1 = \{2^2, 3^3, 4^4, 13^{12}\},$$
$$A^2 = \{1^1, 3^{14}\},$$
$$S^2 = \{1^1, 2^2, 3^3, 4^4, 13^{12}, 3^{14}\},$$
$$A^3 = \{14^{13}\},$$
$$S^3 = \{1^1, 2^2, 3^3, 4^4, 13^{12}, 14^{13}, 3^{14}\},$$
$$A^4 = \{\ \},$$
$$S_{C2} = S^3 + \{15^{15}\}$$
$$= \{1^1, 2^2, 3^3, 4^4, 13^{12}, 14^{13}, 3^{14}, 15^{15}\},$$

and the slice

```
 1 input(n, a);
 2 i := 1;
 3 while i < n do begin
 4    min := a[i];
13    a[i] := min;
14    i := i + 1;
   end;
15 output(a);
```

In contrast to the above example, a static slice [24] derived for a similar slicing criterion $C = (15, \{a\})$ for the program of Fig. 1 is the entire program itself. This illustrates the fact that dynamic slices are, in general, smaller than static ones. There is, however, a price for this advantage: dynamic slice cannot be used to support reasoning about all possible computations w.r.t. a selected set of variables in the program.

5. Conclusions

Although the idea of dynamic program analysis is not new [4,8,11,25], that of dynamic slicing is: It originated during experiments with a recently implemented System for Testing and Debugging (STAD) [14,16]. The main advantage of dynamic slicing is that the size of a slice can be significantly reduced by the identification of those statements in the program that do not have influence on the variables of interest. This is achieved by dynamic analysis based on the program execution trajectory.

Some related works in the area of dependence-based modeling have been reported in the litera-

ture. Most of them deal with dependencies between data items [6]. Additionally, control influence is introduced for constructing program slices [24], for optimization [21], and for static program testing [12]. However, all these models are static, derived from a control flowgraph. The model presented in this paper is dynamic [11] because it is derived mainly from the program execution trajectory.

In the case of arbitrary control flow, the scope of influence can be derived by using the concept of the nearest inverse dominator of test instructions [5,18]. However, for structured programs, the scope of influence can be determined during syntax analysis, as was done in STAD [14].

A promising area of research involves dynamic slicing for pointer variables. Pointers create unique problems since the pointer variable actually represents two variables: the pointer itself and the object pointed to by it. Nameless variables (objects) of a given type are created by calling the standard procedure *new*(*p*) which is, in fact, a dynamic declaration of the object involved: A storage is reserved for an object but no value is assigned to it. It is impossible to identify dynamic objects by means of static analysis. In contrast, by applying a dynamic analysis, a list of dynamic variables might be created and manipulated during program execution. In this manner, it is possible to determine which dynamic objects are pointed to be pointer variables at every point of program execution. It is also possible to determine which objects (dynamic variables) are used or modified at every point of program execution.

References

[1] A.V. Aho and J.D. Ullman, *Principles of Compiler Design* (Addison-Wesley, Reading, MA, 1977).

[2] J.M. Barth, A practical interprocedural data flow analysis algorithm, *Comm. ACM* **21** (9) (1978) 724–736.

[3] J.F. Bergeretti and B.A. Carre, Information-flow and data-flow analysis of **while**-programs, *ACM Trans. Programming Languages & Systems* **7** (1) (1985) 37–61.

[4] F.T. Chan and T.Y. Chen, AIDA—A dynamic data flow anomaly detection system for PASCAL programs, *Software – Practice & Experience* **17** (3) (1987) 227–239.

[5] D.E. Denning and P.J. Denning, Certification of programs for secure information flow, *Comm. ACM* **20** (7) (1977) 504–513.

[6] L.D. Fosdick and L.J. Osterweil, Data flow analysis in software reliability, *Comput. Surveys* **8** (1976) 305–330.

[7] M.S. Hecht, *Flow Analysis of Computer Programs* (North-Holland, Amsterdam, 1977).

[8] J.C. Huang, Detection of data flow anomaly through program instrumentation, *IEEE Trans. Software Engrg.* **SE-5** (3) (1979) 226–236.

[9] C.B. Jones, *Software Development, A Rigorous Approach* (Prentice-Hall, Englewood Cliffs, NJ, 1980).

[10] K. Kennedy, A comparison of two algorithms for global data flow analysis, *SIAM J. Comput.* **5** (1976) 158–180.

[11] B. Korel, *Dependence-Based Modelling in the Automation of the Error Localization in Computer Programs*, Ph.D. Thesis, School of Engineering and Computer Science, Oakland Univ., Rochester, MI, August 1986.

[12] B. Korel, The program dependence graph in static program testing, *Inform. Process. Lett.* **24** (2) (1987) 103–108.

[13] B. Korel and J. Laski, A tool for data flow oriented program testing, *Softfair II, 2nd Conf. on Software Development, Tools, Techniques, and Alternatives*, San Francisco, CA (December 1985) 34–38.

[14] B. Korel and J. Laski, *STAD — A System for Testing and Debugging*, Tech. Rept. TR-CSE-87-08, School of Engineering and Computer Science, Oakland Univ., Rochester, MI, August 1987.

[15] J.W. Laski, A hierarchical approach to program testing, *SIGPLAN Notices* **15** (1980) 77–85.

[16] J. Laski, *Data Flow Testing of Computer Programs*, Tech. Rept. TR-CSE-87-06, School of Engineering and Computer Science, Oakland Univ., Rochester, MI, June 1987.

[17] J.W. Laski and B. Korel, A data flow oriented program testing strategy, *IEEE Trans. Software Engrg.* **SE-9** (3) (1983) 347–354.

[18] T. Lengauer and R.E. Tarjan, A fast algorithm for finding dominators in a flowgraph, *ACM Trans. Programming Languages & Systems* **1** (1979) 121–141.

[19] H.D. Longworth, L.M. Ottenstein and M.R. Smith, The relationship between program complexity and slice complexity during debugging tasks, *10th Internat. Computer Software & Applications Conf. (COMSAQ-86)*, Chicago, IL (October 1986) 383–389.

[20] S.S. Muchnick and N.D. Jones, *Program Flow Analysis: Theory and Applications* (Prentice-Hall, Englewood Cliffs, NJ, 1981).

[21] K.J. Ottenstein and L.M. Ottenstein, The program dependence graph in a software development environment, *ACM SIGPLAN Notices* **19** (5) (1984) 177–184.

[22] B.K. Rosen, Data flow analysis for procedural languages, *J. ACM* **26** (2) (1979) 322–344.

[23] M. Weiser, Programmers use slices when debugging, *Comm. ACM* **25** (1982) 446–452.

[24] M. Weiser, Program slicing, *IEEE Trans. Software Engrg.* **SE-10** (4) (1984) 352–357.

[25] N.H. White and K.H. Bennett, Run-time diagnostic in PASCAL, *Software — Practice & Experience* **15** (4) (1985) 359–367.