数理解析研究所講究録
996巻 1997年 54-65

Program Slicing using Functional Networks

Sebastian Danicic and Mark Harman
Project *Project*,
School of Computing,
University of North London,
Eden Grove, London, N7 8DB.
tel: +44 (0)171 607 2789
fax: +44 (0)171 753 7009
e-mail: m.harmanunl.ac.uk

4^{th} RIMS Workshop in Computing, 22^{nd} - 24^{th} July, 1996

Abstract

Program slicing is a technique for identifying a subprogram from an original program. The subprogram, called a slice, is an executable program which maintains the effect of the original upon a chosen set of variables at some point within the Control Flow Graph (CFG) of the original. The variable set, CFG node pair is called the slicing criterion.

Slices find applications in debugging, testing, parallelisation, re-use, code measurement and program comprehension. These applications exploit the way in which slicing preserves a projection of the original program's semantics, allowing, in some cases, for considerable syntactic simplification.

The paper introduces an approach to the computation of a slightly more general form of a slice, called a simultaneous slice. A simultaneous slice is constructed with respect to a set of conventional slicing criteria. The approach introduced here uses a functional implementation of a network of communicating processes, each process corresponding to a CFG node and each message corresponding to either a variable name or a node identifier.

1 Introduction

Program slicing can be used to extract a sub–program concerned with a sub–domain of the overall computation of a program, *provided* this sub–domain is captured at compile time by the values that the program stores in a subset of its variables at some point of interest during the computation. Definition 1.1 gives an informal definition of a slice. This definition is concerned with the static paradigm, where no information describing the inputs to the program is available.

Definition 1.1 (Static Slice)
A slice of a program p is constructed with respect to a slicing criterion (V, n), where V is a set of variables, and n is a point of interest within p. A slice of p is any program constructed from p by the deletion of zero or more statements, and which has the same effect as p upon the variables in V at n[1].

Figure 1, shows a program (in the left hand box) and a slice constructed from it, with respect to the slicing criterion $(\{\texttt{s}\}, 9)$.

Following Weiser's original definition of this static form of program slice [29, 30, 32], several authors have adapted the slicing concept to the dynamic paradigm [2, 16, 9, 17] (in which the input sequence to the program is fully defined at slice–construction time), and to a quasi–static paradigm [26, 28] (where partial information about the input to the program is available). This paper is concerned solely with the static slicing paradigm.

Because a slice captures the thread of computation pertinent to a chosen set of variables at some point in the original program, it has many applications, all of which are based upon the fact that a slice is a simplified version

[1] It is conventional to number the statements of the program, in order to identify statements of interest.

1 read(n); 2 s:=0; 3 p:=1; 4 while n>0 do 5 begin 6 s:=s+n; 7 p:=p*n; 8 n:=n-1 9 end	1 read(n); 2 s:=0; 3 4 while n>0 do 5 begin 6 s:=s+n; 7 8 n:=n-1 9 end
Original Program	Slice w.r.t. ({s}, 9)

Figure 1: A Pascal Program Fragment and One of its Static Slices

of the original which maintains a projection of its semantics. These applications include cohesion measurement [4, 22, 18, 11], algorithmic debugging [23, 15, 10], re-engineering [19, 24], component re-use [3], automatic parallelisation [31], maintenance and debugging [8, 20] and program integration [13]. Tip [27] and Binkley and Gallagher [5] provide detailed surveys of the paradigms, applications and algorithms for program slicing.

Simultaneous slicing is a generalisation of slicing, where the slicing criterion, rather than being a single point in the program, is a whole set of program point, variable set pairs.

A fixpoint theory for simultaneous slicing is developed in terms of Functional Networks. Functional Networks (FNs) are a structure based on the networks introduced by Kahn[14], in his theory of parallel programs and those used by Abramsky[1] for reasoning about concurrent systems. Directly from this 'semantics of simultaneous slicing', an algorithm for simultaneous slicing is derived.

One of the useful properties of an FN is that is can be viewed both as a set of functions written in a pure functional language or as a network of intercommunicating processes. The composition of functions corresponds to the topology of the network, while the parameter passing between functions corresponds to message passing in the network.

For the problem of slicing, messages will consist of variable names and 'node identifiers', and each function in the FN will represent a node in the CFG of the program to be sliced. The variable names represent variables whose value must be preserved in the slice and the node identifiers represent nodes of a program's CFG which have already been included in the slice.

Functional languages, with their higher order semantics and polymorphic functions on sets, provide a very natural medium for implementing such process communication systems. The language Hope is used for definiteness, although any similar functional notation would be equally suitable.

The rest of this paper is organised as follows. Section 2 presents an overview of the parallel slicing algorithm first introduced in [6] and section 4 introduces Functional Networks. Section 5 presents the slice semantics of FNs, which is used in section 6 to develop an implementation of slicing using the functional language Hope.

2 The Parallel Slicing Algorithm

This section briefly reviews the parallel algorithm introduced in [6].

The algorithm has two distinct stages:

1. A Compilation Stage: The program to be sliced[2] is compiled into a *functional network*.
2. An Execution Stage: The functional network is executed. The statements of the object program to be kept in the final slice are obtained from the resulting input to the process corresponding to the *START* node of the CFG of the program to be sliced.

[2]From now on, the program to be sliced will be referred to as the 'object program'.

2.1 The Compilation Stage

The compilation stage is straight–forward. The topology of the network is that of the object program's Reverse Control Flow Graph (RCFG)[3] [12]. Each arc of the RCFG corresponds to an I/O channel in the process network. Each node of the RCFG corresponds to a process in the network whose behaviour is described by definition 2.1.

2.1.1 Process States

The behaviour of each process depends entirely on its *state*. The state of each process is derived from the object program and consists of a four–tuple containing the following information:

n	The node identifier of process n
REF(n)	The set of referenced variables of n
DEF(n)	The set of defined variables of n
$C(n)$	The set of controlled nodes of n

An algorithm for calculating $C(n)$ is given in [7] and the calculation of REF(n) and DEF(n) is trivial [29].

2.1.2 Process Behaviour

Each process is a process which sends and receives messages, each message being a set of variable names and process identifiers.

Suppose a process n receives a set I. If I has any elements in common with DEF(n) or $C(n)$ then the process n outputs a set consisting of :

1. all elements of I that it does not define,
2. all variables that it references and
3. its own node identifier.

otherwise process n simply copies the input I to its output channels.
More formally, the behaviour of process n is denoted $\delta(n)$ in definition 2.1.

Definition 2.1 (Process Behaviour)

$$\delta(n) = \lambda I. \begin{cases} (I - \text{DEF}(n)) \cup \text{REF}(n) \cup \{n\} & \text{if } I \cap (\text{DEF}(n) \cup C(n)) \neq \emptyset, \\ I & \text{otherwise.} \end{cases}$$

2.2 The Execution Stage

2.2.1 Initiating the Network

In order to construct a slice for the criterion (V, n), network communication is initiated by placing the set V on the output arc(s) of process n.

2.2.2 Changes in the Network State

The network state changes as a result of process outputting values in response to inputs, thereby causing new sets to appear on the arcs of the network.

A process n outputting a set, O, has the effect of updating the value(s) of all the output arc(s) of n by 'unioning' what was on the arc(s) before with O. The state of each process, however, remains unchanged as the communication progresses.

[3] A program's RCFG is simply the inverse of its CFG. Note that an RCFG is itself a CFG. The START node of a program's RCFG is the STOP node of its CFG and vice-versa.

```
1     i=1;
2     s=0;
3     p=1;
4     while(i<=N) {
5        s=s+i;
6        p=p*i;
7        i=i+1; }
8
```

Figure 2: The 'Sum and Product' Program

2.2.3 Termination

Eventually the network reaches a state where further communication causes no new messages to be created on any arc. At this stage communication ceases and the network is said to be stable.

2.3 The Final Result

The statements of an object program that should be included in the final slice are precisely those whose corresponding identifiers are on an input arc of the STOP process when the network stabilises [6].

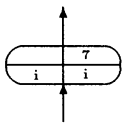
3 An Example

3.1 Example Compilation

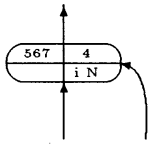
The object program is the 'Sum and Product' Program given in Figure 2.

Diagramatically, the state of each process will be represented by placing the values of each four state components in one of four quadrants. Defined variables will be placed in the lower left quadrant, referenced variables in the lower right, controlled nodes in the upper left and the node identifier in the upper right.

Each process corresponds to one of the numbered 'statements' of the 'Sum and Product' program. Process 7, for example, corresponds to `i:=i+1` and so has DEF(7) = REF(7) = {i} and has $C(7) = \emptyset$. Its state is thus shown as:



Processes corresponding to primitive statements of the object program will have one input channel. Process 4, however, is a called 'predicate process', corresponding to a predicate in the object program and thus has two input channels. Also, being a typical predicate, process 4 controls other processes, in this case, $C(4) = \{5, 6, 7\}$. Process 4 also references the two variables `i` and `N`, and, being side–effect free, has no defined variables. Its state, therefore, is:



The network constructed for the program in Figure 2 is thus as shown in Figure 3.

3.1.1 Example Execution

Let the slicing criterion be ({p}, 8).

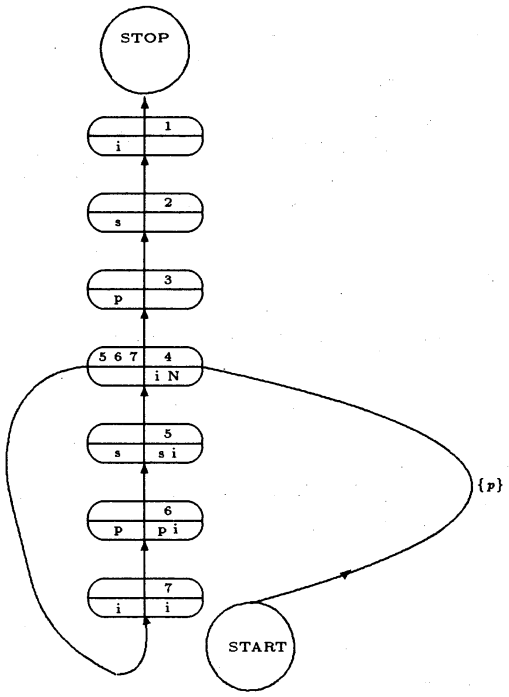


Figure 3: Initial Network State

Execution is initiated by placing {p} on the output channel of the START process.

The set {p} passes through processes 4 and 5 without effect, until it reaches process 6. Since DEF(6) = {p}, this input causes process 6 to output {p, i, 6}. This is an example of a process responding to a *data dependence* in the object program. The set {p, i, 6} passes through process 5 without effect. At the same time, {p} passes from process 4 to process 3, whereupon, process 3 outputs {3} This set passes on through processes 2 and 1 without effect. The network state at this stage is picture in Figure 3.1.1.

Since the $C(4) = \{5, 6, 7\}$, the input {p, i, 6} causes process 4 to output {p, i, N, 6, 4}. This is an example of a process responding to a control dependence in the original program.

The state of the network at this stage is pictured in Figure 5.

The set {p, i, N, 6, 4} having been output from process 4, passes round the cycle causing process 7 to output {p, i, N, 7, 6, 4} also through process 2 and 1 causing process 1 to output {p, i, N, 7, 6, 4, 1}. Once the output from process 7 has reached the STOP process of the network, no new I/O occurs and the system has stabilised in the final network state pictured in Figure 6.

In this state those process identifiers which reside as input arcs to the STOP process is $\{7, 6, 4, 3, 1\}$ and thus the slice for the 'Sum and Product' program produced by this algorithm is as shown in Figure 7.

4 Functional Networks(FNs)

A Functional Network is simply a generalisation of the network described above. Let N be a set of Processes and S a set (of messages). A functional network is a pair (G, δ), consisting of a CFG, G, and a *process function*, δ, the type of which is:

$$\delta : N \rightarrow (S \rightarrow S)$$

where S is a set of subsets consisting of variable names and node identifiers. Importantly, for each particular program, S will be *finite* since it is bounded above by the union of the set of variable names mentioned in the program together with the set of labels (node identifiers) of the program. Observe that S forms a finite lattice with respect to $\subseteq$, with $\perp = \emptyset$, and $\top$ equal to the set consisting of all the variable names and labels (node identifiers) mentioned in the program from which the FN was derived.

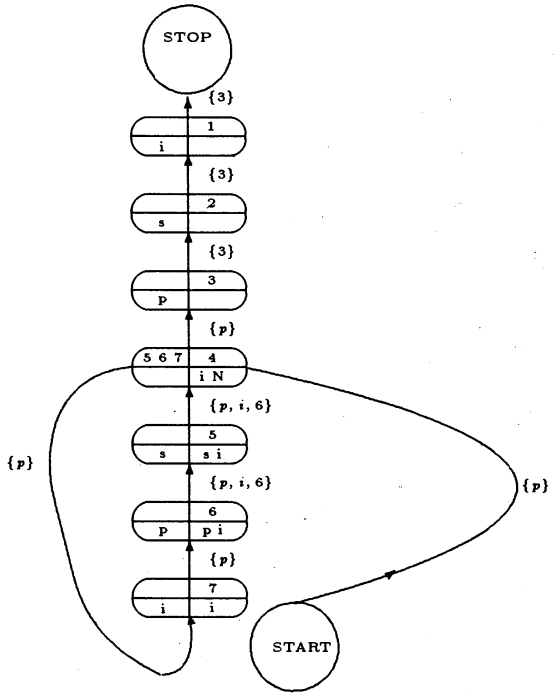


Figure 4: An intermediate Network State

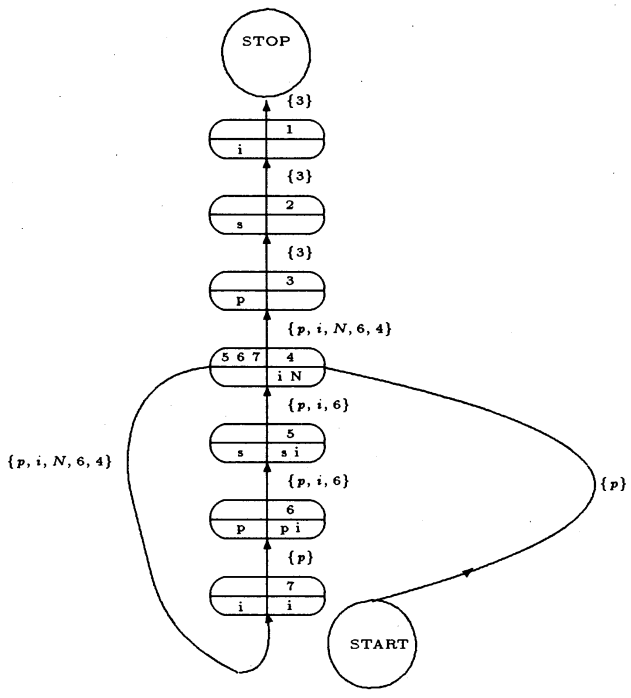


Figure 5: An intermediate Network State

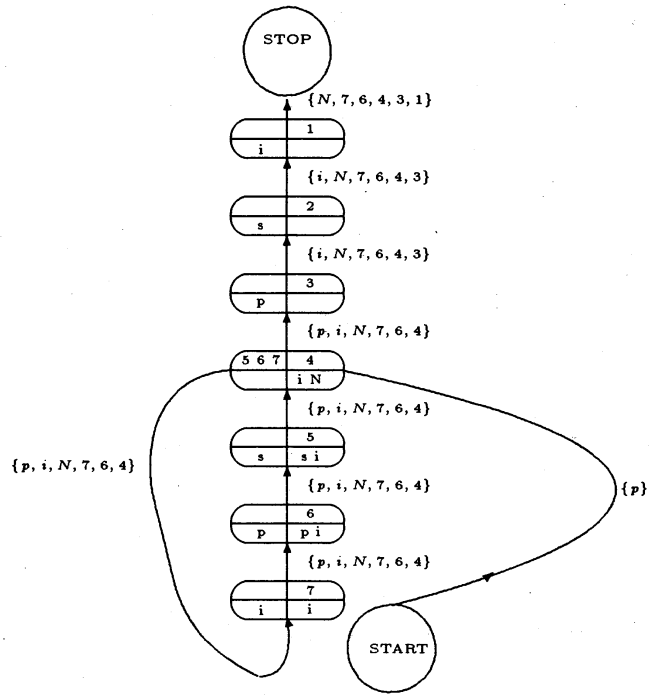


Figure 6: Final Network State

```
1       i=1;
3       p=1;
4       while(i<=N) {
6          p=p*i;
7          i=i+1; }
```

Figure 7: The slice produced with respect to $(\{\mathtt{p}\}, 8)$

Also, essentially, each process function $\delta(n)$ is monotonic, and hence, since S is finite, the $\delta(n)$ are continuous[25] with respect to $\subseteq$. Furthermore, since S is finite, it is a complete lattice and therefore the relevant least upper bounds and least fixpoints all exist.

It will be shown, by appealing to Kleene's First Recursion Theorem[21], that continuity of the $\delta(n)$ guarantees that the required fixpoints corresponding 'best' (i.e. thinnest) slices can be reached in a finite number of steps, as the least upper bounds of Kleene Chains.

Definition 4.1 (A Functional Network (FN))
Let S be a finite lattice of messages,

An FN is a pair (G, δ), consisting of a CFG, G, and a *process function*

$$\delta : N \to (S \to S)$$

such that

$$\text{Processes}(G) \subseteq N$$

and

$$\forall n \in N, \delta(n) \text{ is continuous.}$$

5 The Slice Semantics of FNs

5.1 Simultaneous Slicing Criteria

A *Simultaneous Slicing Criterion* was introduced in section 1. It is simply a labelling of the arcs of G with elements of S. Namely a function L of type:

$$L : G \to S.$$

Note that labelling function L is total. Technically, arcs not mentioned in the slicing criterion will be mapped to the empty set in its corresponding arc labelling.

Definition 5.1 (Slicing Criteria of an FN)
A *slicing criterion* for a FN, (G, δ) is simply a labelling of G.

5.2 Termination

Execution of the network terminates when further activity has no effect on the labelling of all of the arcs of the network. A network reaches this state when its arcs are labelled with what is called a *Stable Labelling*. In this state the labelling function represents a stable network.

Definition 5.2 (A Stable Labelling of an FN)
A *Stable Labelling* of (G, δ) is a function

$$L : G \to S$$

such that

$$(a, b) \in G \text{ and } (a, c) \in G \implies L(a, b) = L(a, c)$$

and

$$(a, b) \in G \text{ and } (b, c) \in G \implies \delta(b)(L(a, b)) \subseteq L(b, c)$$

The first condition simply ensures that all arcs emerging from a process are labelled with the same values. The second condition states that the labelling of the output arcs from each process is a set containing the result of applying the process's corresponding process function to the value of the labels of its input arcs. Since this is true for every process in the network, it is clear that further activity cannot change this network state. The network is thus considered to be *stable*.

5.3 A finite lattice of stable labellings

From any FN, a finite lattice of stable labellings, ordered by pointwise set inclusion is derived. i.e. $L_1 \sqsubseteq L_2 \iff \forall x \in G, L_1(x) \subseteq L_2(x)$.

5.4 A slice of an FN

Given a slicing criterion C, a slice with respect to C is simply a stable labelling 'at least as big' as C.

Definition 5.3 (A Slice of an FN)
Let C be a slicing criterion of (G, δ), and let L be a stable labelling of (G, δ), then L is a *slice* of (G, δ) with respect to C if and only if $C \sqsubseteq L$.

5.5 The 'Best' Slice

Given an FN (G, δ) and a slicing criterion C, the best slice of (G, δ) with respect to C, denoted $\mathcal{BS}_C^{(G,\delta)}$, is the least stable labelling containing C, i.e.

Definition 5.4 (The Best Slice of an FN)

1. $\mathcal{BS}_C^{(G,\delta)}$ is a *slice* of (G, δ) with respect to C
   and
2. L is a slice of (G, δ) with respect to $C \implies \mathcal{BS}_C^{(G,\delta)} \sqsubseteq L$

$\mathcal{BS}_C^{(G,\delta)}$ corresponds to a labelling where the network has stabilised. That is each of the arcs are labelled with sets that are 'as small as possible' but still contain the slicing criterion. Since the set of stable labellings of an FN is a finite lattice, and hence complete, best slices always exist.

The 'Best Slice Theorem' that follows defines a function $f_{(G,\delta)}$ on slicing criteria for (G, δ) whose least fixpoint containing C must be the best slice of (G, δ) with respect to C. Intuitively, applying the function $f_{(G,\delta)}$ corresponds to executing a single 'click' or 'ripple' of the parallel slicing algorithm introduced in [6].

Because $f_{(G,\delta)}$ is continuous, Kleene's First Recursion Theorem guarantees that successive applications of $f_{(G,\delta)}$ will take the network into a state corresponding to the best slice. The finiteness of S guarantees that this process will terminate.

Theorem 5.1 (The Best Slice Theorem)
Let $f_{(G,\delta)} : ((G \to S) \to (G \to S))$ be defined as

$$f_{(G,\delta)} = \lambda C.\lambda(x, y).C((x, y)) \cup \bigcup_{(z,x) \in G \triangleright x} \delta(x)(C((z, x)))$$

where $G \triangleright x$ is the set of all arcs of G which end in x i.e. $\{(y, z) \in G \mid z = x\}$ then
$\mathcal{BS}_C^{(G,\delta)}$ = the least fixpoint of $f_{(G,\delta)}$ containing $C = \bigsqcup_i f^i_{(G,\delta)}(C)$.

Proof

Clearly z is a stable labelling of $(G, \delta) \iff z$ is a fixpoint of $f_{(G,\delta)}$.
So z is a slice of (G, δ) w.r.t $C \iff z$ is a fixpoint of $f_{(G,\delta)}$ and $C \sqsubseteq z$.
So $\mathcal{BS}_C^{(G,\delta)}$ is the least fixpoint of $f_{(G,\delta)}$ containing C.
But $\bigsqcup_i f^i_{(G,\delta)}(C)$. is a fixpoint $f_{(G,\delta)}$ containing C.
Moreover, suppose B is a fixpoint $f_{(G,\delta)}$ containing C,
then $f^i_{(G,\delta)}(C) \sqsubseteq B$ for all i by induction,
therefore $\bigsqcup_i f^i_{(G,\delta)}(C) \sqsubseteq B$.

```
dec mapset: (alpha -> set (beta)) X set(alpha) -> set(beta);
--- mapset(f,{}) <= {};
--- mapset(f,s) <= let (x,y) == split(s)
                   in f(x) union mapset(f,y);

type node == num;
data message == nn(node) ++ vn(list(char));
type S == set(message);
type N == node;
dec eq: (num -> alpha)  X (num -> alpha)  X num -> truval;
--- eq(f,g,0) <= f(0) = g(0);
--- eq(f,g,succ(n)) <= if f(succ(n)) = g(succ(n))
                       then eq(f,g,n)
                       else false;

dec BS: (num X (N->S) X (N->(S->S)) X (N->set(N))) -> (N->S);
--- BS(max,C,delta,G) <=
let fGdelta == lambda x =>
               C(x) union (delta(x))mapset(C,G(x))
in if eq(C,fGdelta,max)
   then C
   else BS(max,fGdelta,delta,G);
```

Figure 8: Hope Implementation of the Slicing Algorithm

So $\bigsqcup_i f^i_{(G,\delta)}(C)$ is the least fixpoint of $f_{(G,\delta)}$ that contains C.

So $\mathcal{BS}^{(G,\delta)}_C$ is the least fix point of $f_{(G,\delta)}$ containing $C = \bigsqcup_i f^i_{(G,\delta)}(C)$ as required.

6 Implementing Static Slicing Using FNs

The theory above has shown that in order to implement an algorithm to produce a best slice, all that is required is to implement the process of producing successive approximations to $\mathcal{BS}^{(G,\delta)}_C$ by repeated applications of $f_{(G,\delta)}$. The implementation is defined in the functional language Hope in Figure 8.

7 Conclusion and Future Work

This paper introduces a functional implementation of the concurrent static slicing algorithm introduced in [6]. The implementation relies upon a theory of functional networks, which can be viewed in either of two ways — as a composition of functions on sets of messages or as a network of communicating processes. The theory guarantees the existence of 'best slices' (slices which are as thin as possible given the approximate nature of the calculation of defined and referenced variables, and the simplifying assumption that no define–use chains are infeasible). The guarantee, in turn, ensures that the implementation of the functional network (as a Hope program) terminates, yielding the 'best slice'.

Work is currently in progress to extend this theory and its associated implementation to the problem of interprocedural slicing (slicing across procedure boundaries) and to handle slicing in the presence of pointer aliasing.

The authors also believe that the close correspondence between a CFG of a program and the concurrent solution to graph theoretic problems involving the CFG may be more widely applicable.

References

[1] ABRAMSKY, S. Reasoning about concurrent systems. In *Distributed Computing* (London, 1984), pp. 307–319.

[2] AGRAWAL, H., AND HORGAN, J. R. Dynamic program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, June 1990), pp. 246–256.

[3] BECK, J., AND EICHMANN, D. Program and interface slicing for reverse engineering. In *IEEE/ACM* 15^{th} *Conference on Software Engineering (ICSE'93)* (1993), pp. 509–518.

[4] BIEMAN, J. M., AND OTT, L. M. Measuring functional cohesion. *IEEE Transactions on Software Engineering 20*, 8 (Aug. 1994), 644–657.

[5] BINKLEY, D. W., AND GALLAGHER, K. B. Program slicing. *Annals of Computing* (1996). To appear.

[6] DANICIC, S., HARMAN, M., AND SIVAGURUNATHAN, Y. A parallel algorithm for static program slicing. *Information Processing Letters 56*, 6 (Dec. 1995), 307–313.

[7] FERRANTE, J., OTTENSTEIN, K. J., AND WARREN, J. D. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems 9* (July 1987), 319–349.

[8] GALLAGHER, K. B., AND LYLE, J. R. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering 17*, 8 (Aug. 1991), 751–761.

[9] GOPAL, R. Dynamic program slicing based on dependence graphs. In *IEEE Conference on Software Maintenance* (1991), pp. 191–200.

[10] HARMAN, M., AND DANICIC, S. Using program slicing to simplify testing. *Journal of Software Testing, Verification and Reliability 5* (Sept. 1995), 143–162.

[11] HARMAN, M., DANICIC, S., SIVAGURUNATHAN, B., JONES, B., AND SIVAGURUNATHAN, Y. Cohesion metrics. In 8^{th} *International Quality Week* (San Francisco, May 29th – June 2nd. 1995), pp. Paper 3–T–2, pp 1–14.

[12] HECHT, M. S. *Flow Analysis of Computer Programs.* Elsevier, 1977.

[13] HORWITZ, S., PRINS, J., AND REPS, T. Integrating non–interfering versions of programs. *ACM Transactions on Programming Languages and Systems 11*, 3 (July 1989), 345–387.

[14] KAHN, G. A simple theory of parallel programs. In *International Foundations of Information Processing, World Congress* (1974).

[15] KAMKAR, M. *Interprocedural dynamic slicing with applications to debugging and testing.* PhD Thesis, Department of Computer Science and Information Science, Linköping University, Sweden, 1993. Available as Linköping Studies in Science and Technology, Dissertations, Number 297.

[16] KAMKAR, M., SHAHMEHRI, N., AND FRITZSON, P. Interprocedural dynamic slicing. In *Proceedings of the* 4^{th} *Conference on Programming Language Implementation and Logic Programming* (1992), pp. 370–384.

[17] KOREL, B., AND LASKI, J. Dynamic program slicing. *Information Processing Letters 29*, 3 (Oct. 1988), 155–163.

[18] LAKHOTIA, A. Rule–based approach to computing module cohesion. In *Proceedings of the* 15^{th} *Conference on Software Engineering (ICSE-15)* (1993), pp. 34–44.

[19] LIU, L., AND ELLIS, R. An approach to eliminating COMMON blocks and deriving ADTs from Fortran programs. Technical report, University of Westminster, UK, Feb. 1993.

[20] LYLE, J. R., AND WEISER, M. Automatic program bug location by program slicing. In 2^{nd} *International Conference on Computers and Applications* (Peking, 1987), pp. 877–882.

[21] MANNA, Z. *Mathematical Theory of Computation.* McGraw–Hill, 1974.

[22] OTT, L. M., AND THUSS, J. J. Slice based metrics for estimating cohesion. In *Proceedings of the IEEE-CS International Metrics Symposium* (1993), pp. 78–81.

[23] SHAHMEHRI, N. *Generalized algorithmic debugging.* PhD Thesis, Department of Computer Science and Information Science, Linköping University, Sweden, 1991. Available as Linköping Studies in Science and Technology, Dissertations, Number 260.

[24] SIMPSON, D., VALENTINE, S. H., MITCHELL, R., LIU, L., AND ELLIS, R. Recoup – Maintaining Fortran. *ACM Fortran forum 12*, 3 (Sept. 1993), 26–32.

[25] STOY, J. E. *Denotational semantics: The Scott–Strachey approach to programming language theory.* MIT Press, 1985. Third edition.

[26] TIP, F. *Generation of Program Analysis Tools.* PhD thesis, Centrum voor Wiskunde en Informatica, Amsterdam, 1995.

[27] TIP, F. A survey of program slicing techniques. *Journal of Programming Languages* (1995). To appear.

[28] VENKATESH, G. A. The semantic approach to program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (Toronto, Canada, June 1991), pp. 26–28. Proceedings in *SIGPLAN Notices*, 26(6), pp.107–119, 1991.

[29] WEISER, M. *Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method.* PhD thesis, University of Michigan, Ann Arbor, MI, 1979.

[30] WEISER, M. Program slicing. In 5^{th} *International Conference on Software Engineering* (San Diego, CA, Mar. 1981), pp. 439–449.

[31] WEISER, M. Reconstructing sequential behaviour from parallel behaviour projections. *Information Processing Letters 17*, 10 (1983), 129–135.

[32] WEISER, M. Program slicing. *IEEE Transactions on Software Engineering 10*, 4 (1984), 352–357.