Postprint of article in *Information Processing Letters* **25** (3): 189–193 (1987)

On the Detection of Unstructuredness in Flowgraphs *

T.H. Tse
Department of Computer Science
The University of Hong Kong
Pokfulam
Hong Kong

Keywords: Structured programming, flowgraphs.

CR Categories: D.2.2, G.2.2

1. Background

Although structured programming has been introduced for about two decades, there are still a lot of unstructured programs which need to be maintained [6]. Over 75 per cent of the costs in a system life cycle are spent on maintenance [4]. Because of this, quite a number of papers have discussed the problems of detecting program unstructuredness and proposed numerous ways of restructuring them. These papers can be divided into three categories:

(a) Most of the papers are based on heuristic or intuitive arguments [2, 5, 6, 10, 11, 12, 17, 20, 21, 22]. Each of the papers tries to propose a new method which is supposedly better than the previous ones [13]. According to the authors, unstructuredness is caused by four elements –– branching into a selection, branching out of a selection, branching into a loop and branching out of a loop. The detection of these unstructured elements is a difficult task. Since none of these unstructured elements can exist in isolation, the authors recommend that we should test unstructured compounds, which are combination of unstructured elements. Unfortunately, the number of combinations is endless [19], so that this approach cannot be exhaustive.

(b) The second approach uses the concept of reducibility of program schemes [7, 8, 9, 13, 18]. In essence, a program scheme M_1 is reducible to another scheme M_2 if M_1 can be transformed to M_2 according to some defined rules. A program is structured if it can be reduced to a structured scheme, consisting only of sequences, selections and while-loops. It is not structured if irreducible forms result. But in practice, the number of irreducible forms increases exponentially with the number of decisions in a program [7], and hence it is very difficult to decide on the reducibility of large and complex programs.

(c) A third approach uses the concept of succession paths in flowgraphs to define the minimum module related with a given node. It seems to be the most promising because it produces simple but working algorithms. Urschler [16], for example, proposed an efficient algorithm for structuring programs automatically. However, since no attempt was made to isolate unstructuredness, a lot of time would be wasted in restructuring programs which were already structured. Becerril *et al.* [3] attempted both to supplement the approach with a theoretical basis and to discuss the conditions for unstructuredness. Unfortunately, no less than nine non-trivial errors were introduced before the conditions for unstructuredness were proposed. The results claimed in the paper were therefore unreliable.

* Part of this research was done at the London School of Economics, University of London under a Commonwealth Academic Staff Scholarship. It was also supported in part by a University of Hong Kong Research Grant.

Because of the problems of the first two methods and the promise of the third, we shall study the conditions for unstructuredness in terms of succession paths. We shall formally define multiple entries and exits, and show that two simple conditions are sufficient for the detection of unstructuredness. Only an outline of the theory will be given in this paper. Interested readers may refer to [15] for further details.

2. Skeletons

In this section, we shall define some of the properties of flowgraphs through the concept of succession paths.

We define a *flowgraph* as a finite set of nodes, together with two successor functions $s_{\mathbf{true}}$ and $s_{\mathbf{false}}$ defined on the nodes, satisfying four conditions:

(a) There is a unique node, denoted by **begin**, such that $s_\alpha(n) \neq \mathbf{begin}$ for any node n and any Boolean* value α.

(b) For any node n other than **begin**, there exists a finite sequence of nodes $<\mathbf{begin}\ m_0 \ldots\ m_r>$, together with a finite sequence of Boolean values $<\beta_0 \ldots\ \beta_{r-1}>$, such that

$m_0 = s_{\mathbf{true}}(\mathbf{begin})$;
$m_i = s_{\beta_{i-1}}(m_{i-1})$ for $i = 1 \ldots\ r$ (if $r > 0$);
$m_r = n$.

This sequence is known as an *ancestry path*.

(c) There is a unique node, denoted by **end**, such that

$s_{\mathbf{true}}(\mathbf{end}) = s_{\mathbf{false}}(\mathbf{end}) = \mathbf{end}$.

(d) For any node n other than **end**, there exists a finite sequence of nodes $<m_0 \ldots\ m_r\ \mathbf{end}>$, together with a finite sequence of Boolean values $<\beta_0 \ldots\ \beta_r>$, such that

$m_0 = n$;
$m_i = s_{\beta_{i-1}}(m_{i-1})$ for $i = 1 \ldots\ r$ (if $r > 0$);
$\mathbf{end} = s_{\beta_r}(m_r)$.

This sequence is known as a *succession path*.

Our concept of succession is equivalent to the concepts of dominance of [13], back dominance of [8], reverse dominance of [18], postdominance of [3, 16] and descendants of [14]. The term ‘‘succession’’ is chosen to avoid the controversy towards the meaning of the word ‘‘dominance’’.

We divide the nodes in a flowgraph into two classes: *action nodes* and *decision nodes*. For any action node n, $s_{\mathbf{true}}(n) = s_{\mathbf{false}}(n)$. That is to say, an action node has a unique successor. On the other hand, for any decision node n, $s_{\mathbf{true}}(n) \neq s_{\mathbf{false}}(n)$. For completeness, we shall treat the node **begin** as a decision node by defining $s_{\mathbf{false}}(\mathbf{begin}) = \mathbf{end}$.

A succession path is said to be *elementary* if it does not contain more than one occurrences of the same node. A node m is known as a *common successor* of another node n if all the elementary succession paths $<s_{\mathbf{true}}(n) \ldots\ \mathbf{end}>$ and $<s_{\mathbf{false}}(n) \ldots\ \mathbf{end}>$ contain m. In this case, we write $n < m$.

---

* We shall use the Greek letters α and β to denote Boolean values, and $-\alpha$ and $-\beta$ to denote their negations.

We define the *least common successor* of n, denoted by $s_{\mathsf{V}}(n)$, as a node such that

(a) $n < s_{\mathsf{V}}(n)$;

(b) For any node m, $n < m$ implies $s_{\mathsf{V}}(n) < m$ or $s_{\mathsf{V}}(n) = m$.

Clearly, the least common successor of an action node is its successor and the least common successor of **begin** is **end**. The least common successor of any node in general can be found using the algorithm of [1] or [14].

Given a node n in any flowgraph, we define a *skeleton* $\mathbf{q}_{\alpha}(n)$ as the sequence of nodes $<p_0 \ldots p_r>$ such that

$p_0 = s_{\alpha}(n)$;
$p_i = s_{\mathsf{V}}(p_{i-1})$ for $i = 1 \ldots r$ (if $r > 0$);
$s_{\mathsf{V}}(p_r) = s_{\mathsf{V}}(n)$.

A skeleton is equivalent to a g-chain in [3].

Clearly the skeletons of action nodes are empty. What about decision nodes? Given a decision node n, if one of its skeletons $\mathbf{q}_{\alpha}(n)$ is non-empty, we can show that an elementary succession path $<n\ s_{-\alpha}(n) \ldots$ **end**$>$ always exists. Furthermore, the elementary succession path $<s_{-\alpha}(n) \ldots$ **end**$>$ must contain all the elements of $\mathbf{q}_{-\alpha}(n)$. Based on these two premises, we can arrive at the following proposition:

> **Proposition 1**
> If a decision node n is in one of its own skeletons $\mathbf{q}_{\beta}(n)$, then the opposite skeleton $\mathbf{q}_{-\beta}(n)$ must be empty.

3. Modules

For any node n, we define the *minimal module* containing n (or simply the *module* M_n) as the minimum set of nodes satisfying the following conditions:

(a) n is in M_n;

(b) If m is a decision node in M_n, then all the nodes in both the skeletons $\mathbf{q}_{\mathbf{true}}(m)$ and $\mathbf{q}_{\mathbf{false}}(m)$ are in M_n.

It can be shown that this definition is equivalent to the more complicated definition of modules in [3].

We want to find a condition for deciding whether a node lies inside a given module. Let p be a node in M_n. Then there is a decision node m_0 in M_n such that p lies in one of the skeletons $\mathbf{q}_{\beta_0}(m_0)$. If $m_0 \neq n$, there is another decision node m_1 in M_n such that m_0 lies in $\mathbf{q}_{\beta_1}(m_1)$. Since there are only a finite number of decision nodes, we can reach n by applying the above procedure a finite number of times. Thus, there is a finite sequence of distinct skeletons $<\mathbf{q}_{\beta_r}(m_r) \ldots \mathbf{q}_{\beta_0}(m_0)>$ such that

$p \in \mathbf{q}_{\beta_0}(m_0)$;
$m_{i-1} \in \mathbf{q}_{\beta_i}(m_i)$ for $i = 1 \ldots r$ (if $r > 0$);
$m_r = n$.

Conversely, suppose there is a finite sequence of distinct skeletons $<\mathbf{q}_{\beta_r}(m_r) \ldots \mathbf{q}_{\beta_0}(m_0)>$ satisfying the above three conditions. Since $m_{r-1} \in \mathbf{q}_{\beta_r}(m)$ and $m_r \in M_n$, $m_{r-1} \in M_n$. Proceeding in this way, after a finite number of steps, we can conclude that p also lies in M_n. Hence we obtain the following proposition:

Proposition 2
Given a decision node n, a node p is in the module M_n if and only if there is a finite sequence of distinct skeletons $<\mathbf{q}_{\beta_0}(m_0) \ldots \mathbf{q}_{\beta_r}(m_r)>$ such that

$p \in \mathbf{q}_{\beta_0}(m_0)$;
$m_{i-1} \in \mathbf{q}_{\beta_i}(m_i)$ for $i = 1 \ldots r$ (if $r > 0$);
$m_r = n$.

Similarly, given a decision node n and a Boolean value α, we define a *branch* $B_\alpha(n)$ as the minimum set of nodes satisfying the following conditions:

(a) All the nodes in the

(b) If m is a decision node in $B_\alpha(n)$, then all the nodes in both the skeletons $\mathbf{q}_{\mathbf{true}}(m)$ and $\mathbf{q}_{\mathbf{false}}(m)$ are in $B_\alpha(n)$.

This definition of branches avoids the erroneous concept of ''heads'' in [3].

The next proposition is useful for deciding whether or not a node appears in a given branch.

Proposition 3
Given a decision node n, a node p is in the branch $B_\alpha(n)$ if and only if there is a finite sequence of distinct skeletons $<\mathbf{q}_{\beta_0}(m_0) \ldots \mathbf{q}_{\beta_r}(m_r)>$ such that

$p \in \mathbf{q}_{\beta_0}(m_0)$;
$m_{i-1} \in \mathbf{q}_{\beta_i}(m_i)$ for $i = 1 \ldots r$ (if $r > 0$);
$m_r = n$;
$\beta_r = \alpha$.

Given a node m inside the module M_n, the module M_m is obviously a subset of M_n. But we are also interested in finding a condition for which $M_m = M_n$. We note that, if a node p is in the branch $B_\alpha(m)$, then the module M_p must be a subset of $B_\alpha(m)$. The following proposition and corollary will therefore follow:

Proposition 4
Given a node m in the module M_n, if n is in one of the branches $B_\alpha(m)$, then $M_m = M_n$.

Corollary
Given a node m in the module M_n, if n is in one of the skeletons $\mathbf{q}_\alpha(m)$, then $M_m = M_n$.

4. Unstructuredness

Intuitively, unstructuredness in flowgraphs is due to exits in the middle of selections, multiple exits in iterations and/or multiple entries to iterations or selections. But it can be shown that exits in the middle of selections will only take place in the presence of multiple exits in iterations or multiple entries. In this section, we shall formally define unstructuredness in flowgraphs through the last two concepts only.

A decision node m is defined as an *iteration exit* of the module M_n if:

(a) $M_m = M_n$;

(b) m is in one of the branches $B_\alpha(m)$ but not in the opposite branch $B_{-\alpha}(m)$.

A module is said to have *multiple iteration exits* if it has more than one iteration exits.

A node m in the module M_n is defined as an entry node of M_n if there is some node p outside M_{0u} such that $m = s_\beta(p)$. A module is said to have *multiple entries* if it has more than one entry nodes.

A module is said to be *unstructured* if it contains multiple iteration exits or multiple entries.

5. DETECTION OF MULTIPLE ITERATION EXITS

In this section we shall find the criterion for detecting multiple iteration exits.

Suppose a node n has two non-empty skeletons $\mathbf{q}_\alpha(n)$ and $\mathbf{q}_{-\alpha}(n)$, and suppose $n \in B_\alpha(n)$. An elementary succession path $<n\ s_\alpha(n)$ ... **end**> will always exist, and we can show that there is a second decision node $m \in B_\alpha(n)$ such that m is an iteration exit of M_n.

Consider also the opposite branch $B_{-\alpha}(n)$. If $n \in B_{-\alpha}(n)$, then there is another iteration exit $p \in B_{-\alpha}(n)$. If, on the other hand, $n \notin B_{-\alpha}(n)$, then n itself is an iteration exit. In either case, the module M_n will have multiple iteration exits. Hence we have the following lemma:

> **Lemma 5**
> If a node n is the only iteration exit in its module M_n, then one of the skeletons $\mathbf{q}_\beta(n)$ must be empty.

Suppose n is the only iteration exit in M_n. Then $n \in B_\alpha(n)$ but $\notin B_{-\alpha}(n)$. Furthermore, by Lemma 5, $B_{-\alpha}(n)$ is empty. Hence $n \in \mathbf{q}_\beta(m)$ for some $m \in B_\alpha(n)$. By the Corollary of Proposition 4, $M_m = M_n$. Assume that $m \neq n$. Since n is the only iteration exit, m must be in both $B_\beta(m)$ and $B_{-\beta}(m)$. Therefore $B_\beta(m)$ is non-empty, and we can show that an elementary path $<m\ s_{-\beta}(m)$ ... **end**> always exists. We can further show that there is a decision node $p \in B_{-\beta}(m)$ such that p is an iteration exit. But since n is the only iteration exit, p and n must be the same node. In other words, all elementary paths $<m$, $s_{-\beta}(m)$ ... **end**> pass through n. This contradicts the fact that $n \in \mathbf{q}_\beta(m)$. Hence $m = n$, and we can arrive at the following theorem:

> **Theorem 6**
> If a node n is the only iteration exit in the module M_n, then one of its skeletons $\mathbf{q}_\alpha(n)$ must contain n.

Is the converse of the theorem also true? Suppose $\mathbf{q}_\alpha(n)$ contains n. By Proposition 1, $\mathbf{q}_{-\alpha}(n)$ is empty. n is therefore an iteration exit. Assume that there is another iteration exit m. It must be in $B_\alpha(n)$ or $B_{-\alpha}(n)$. We shall show that we have a contradiction in either case. If $m \in B_\alpha(n)$, then there is a node $p\ (\neq n) \in \mathbf{q}_\alpha(n)$ such that $m \in M_p$. Since both p and n are in $\mathbf{q}_\alpha(n)$, we must have $s_\mathsf{v}(p) < s_\mathsf{v}(n)$. Hence $s_\mathsf{v}(m) < s_\mathsf{v}(n)$, contradicting the fact that $M_m = M_n$. On the other hand, if $m \in B_{-\alpha}(n)$, then, since $n \in M_m$, $n \in B_{-\alpha}(n)$. This contradicts the fact that n is an iteration exit. In short, there cannot be any iteration exit other than n. Thus, we have the following theorem:

> **Theorem 7**
> A node n is the only iteration exit of the module M_n if and only if one of its skeletons $\mathbf{q}_\alpha(n)$ contains n.

Its corollary provides a sufficient and necessary condition for the detection of multiple iteration exits, thus:

Corollary
A module M_n have multiple iteration exits if and only if the node n is in one of the branches $B_\alpha(n)$ but not the corresponding skeleton $\mathbf{q}_\alpha(n)$.

6. Detection of Multiple Entries

The detection of multiple entries is more straightforward. The following theorem is obvious.

Theorem 8
A module M_n has multiple entries if and only if there are two nodes m_1 and m_2 in M_n such that $m_1 \in \mathbf{q}_{\beta_1}(p_1)$, $m_2 \in \mathbf{q}_{\beta_2}(p_2)$ and $m_2 \notin \mathbf{q}_{\beta_1}(p_1)$ for some nodes p_1 and p_2 outside M_n.

7. Conclusion

We have defined the concepts of skeleton, module, branch, entry and iteration exit in a flowgraph. We have shown that two simple conditions are sufficient for the detection of unstructuredness. Namely, a module M_n will be unstructured if:

(a) the node n is in one of its branches $B_\alpha(n)$ but not in the corresponding skeleton $\mathbf{q}_\alpha(n)$, or

(b) there are two nodes m_1 and m_2 in M_n such that $m_1 \in \mathbf{q}_{\beta_1}(p_1)$, $m_2 \in \mathbf{q}_{\beta_2}(p_2)$ and $m_2 \notin \mathbf{q}_{\beta_1}(p_1)$ for some nodes p_1 and p_2 outside M_n.

Acknowledgement

The author is grateful to Ronald Stamper of the London School of Economics, University of London for his invaluable suggestions on the project.

References

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, MA, 1974.

[2] E. Ashcroft and Z. Manna, The translation of 'goto' programs to 'while' programs, in: Classics in Software Engineering, E. Yourdon (ed.), Yourdon Press Computing Series, Prentice Hall, Englewood Cliffs, NJ, 1979, pp. 51−61.

[3] J.L. Becerril, J. Bondia, R. Casajuana, and F. Valer, Grammar characterization of flowgraphs, IBM Journal of Research and Development 24 (6) (1980) 756−763.

[4] B.W. Boehm, Software Engineering Economics, Prentice Hall, Englewood Cliffs, NJ, 1981.

[5] C. Boehm and G. Jacopini, Flow diagrams, Turing machines, and languages with only 2 formation rules, Communications of the ACM 9 (1966) 366−371.

[6] M.A. Colter, Techniques for understanding unstructured code, in: Proceedings of the 6th International Conference on Information Systems, L. Gallegos, R. Welke, and J. Wetherbe (eds.), Indianapolis, IN, 1985, pp. 70−88.

[7] D.F. Cowell, D.F. Gilles, and A.A. Kaposi, Synthesis and structural analysis of abstract programs, The Computer Journal 23 (3) (1980) 243−247.

[8] S.A. Greibach, Theory of Program Structures: Schemes, Semantics, Verification, Lecture Notes in Computer Science 36, Springer, Berlin, Germany, 1975.

[9] S.R. Kosaraju, Analysis of structured programs, Journal of Computer and System Sciences 9 (1974) 232−255.

[10] T.J. McCabe, A complexity measure, IEEE Transactions on Software Engineering 2 (4) (1976) 308−320.

[11] H.D. Mills, Mathematical foundations for structured programming, in: Writings of the Revolution: Selected Readings on Software Engineering, E. Yourdon (ed.), Yourdon Press Computing Series, Prentice Hall, Englewood Cliffs, NJ, 1982, pp. 220−262.

[12] G. Oulsman, Unraveling unstructured programs, The Computer Journal 25 (3) (1982) 379−387.

[13] R.E. Prather and S.G. Giulieri, Decomposition of flowchart schemata, The Computer Journal 24 (3) (1981) 258−262.

[14] R. Tarjan, Depth first search and linear graph algorithms, SIAM Journal on Computing 1 (2) (1972) 146−160.

[15] T.H. Tse, The identification of program unstructuredness: A formal approach, The Computer Journal 30 (6) (1987) 507−511.

[16] G. Urschler, Automatic structuring of programs, IBM Journal of Research and Development 19 (1975) 181−194.

[17] M.H. Williams, Generating structured flow diagrams: The nature of unstructuredness, The Computer Journal 20 (1) (1977) 45−50.

[18] M.H. Williams, A comment on the decomposition of flowchart schemata, The Computer Journal 25 (3) (1982) 393−396.

[19] M.H. Williams, Flowchart schemata and the problem of nomenclature, The Computer Journal 26 (3) (1983) 270−276.

[20] M.H. Williams and G. Chen, Restructuring Pascal programs containing goto statements, The Computer Journal 28 (2) (1985) 134−137.

[21] M.H. Williams and H.L. Ossher, Conversion of unstructured flow diagrams to structured form, The Computer Journal 21 (2) (1978) 161−167.

[22] E. Yourdon, Techniques of Program Structure and Design, Yourdon Press Computing Series, Prentice Hall, Englewood Cliffs, NJ, 1975.