HKU CSIS Tech Report TR-C4-86

Postprint of article in *The Computer Journal* **30** (6): 507−511 (1987)

# The Identification of Program Unstructuredness: a Formal Approach*

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

**ABSTRACT**

Quite a number of papers have discussed the problems of identifying unstructured programs and turning them into structured programs. Most of the papers, however, are based on intuitive arguments rather than on formal proofs. The identification of program unstructuredness has remained a difficult task. In this paper we formally study the properties of skeletons, modules, branches, iteration exits and entry nodes in program flowgraphs. We prove that two simple conditions are sufficient and necessary for the identification of unstructuredness.

## 1. INTRODUCTION

The identification of program unstructuredness is important for two reasons:

(a) Although the concept of structured programming has been around for many years, there are still a lot of unstructured programs which need to be maintained [4]. Over three-quarters of the cost in a system life cycle is spent on maintenance [3].

(b) The author is interested in developing a theoretical model which turns a flow diagram into a program structure chart [12, 14, 15, 13]. Since flow diagrams are mainly used as a communication tool with users during the analysis stage of system development, they are problem-oriented and may not necessarily be structured. An algorithm must be available to detect any unstructuredness in the flow diagrams before structure charts can be produced.

Quite a number of papers [1, 2, 4, 5, 6, 7, 8, 9, 10, 16, 17, 18, 19, 20, 21] have discussed the problems of identifying unstructuredness in programs and turning them into structured programs. An excellent summary of the approaches can be found in Williams [19]. Most of the papers, however, are based on intuitive arguments rather than on formal proofs. Even papers with formal proofs (such as Becerril et al. [2] and McCabe [7]) contain errors which make the results untrustworthy. As pointed out by Williams [19] and Oulsman [10], the identification of program unstructuredness has remained a difficult task. In this paper, we shall formally study the properties of skeletons, modules, branches, iteration exits and entry nodes in program flowgraphs. We shall prove that only two simple conditions are sufficient and necessary for the identification of unstructuredness. Only standard set-theoretic results will be used in the proofs. Theorems in graph theory or partially ordered sets will not be assumed.

* A preliminary version of this paper, entitled "On the detection of unstructuredness in flowgraphs", was published in *Information Processing Letters* **25** (3): 189−193 (1987). Part of this research was done at the London School of Economics, University of London under a Commonwealth Academic Staff Scholarship.

## 2. PROGRAM FLOWGRAPHS AND SKELETONS

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

(a) There exists one and only one node, denoted by **begin**, such that $s_\alpha(n) \neq$ **begin** for any node $n$ and any Boolean* value $\alpha$.

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

$m_0 = s_{\mathbf{true}}(\mathbf{begin})$;

$m_i = s_{\beta_{i-1}}(m_{i-1})$ for $i = 1, ..., r$ (if $r > 0$);

$m_r = n$.

This sequence is known as an *ancestry path*.

(c) There exists one and only one 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 <$m_0$, ..., $m_r$, **end**> and a finite sequence of Boolean values <$\beta_0$, ..., $\beta_r$> such that

$m_0 = n$;

$m_i = s_{\beta_{i-1}}(m_{i-1})$ for $i = 1, ..., r$ (if $r > 0$);

$\mathbf{end} = s_{\beta_r}(m_r)$.

This sequence is known as a *succession path*.

The nodes in a program flowgraph can be divided into two categories: *action node* and *decision node*. For any action node $n$, $s_{\mathbf{true}}(n) = s_{\mathbf{false}}(n)$. In other words, an action node has only one successor. On the other hand, for any decision node $n$, $s_{\mathbf{true}}(n) \neq s_{\mathbf{false}}(n)$. For the sake of convenience, we shall regard the **begin** node 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 <$s_{\mathbf{true}}(n)$, ..., **end**> and <$s_{\mathbf{false}}(n)$, ..., **end**> contain $m$. In this case we write $n < m$.

The *least common successor* of $n$, denoted by $s_\vee(n)$, is defined as a node ($\neq n$) such that

(a) $n < s_\vee(n)$;

(b) For any node $m$, $n < m$ implies $s_\vee(n) < m$ or $s_\vee(n) = m$.**

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 [11].

* Greek letters $\alpha$, $\beta$ and so on will be used throughout the paper to denote Boolean values. $-\alpha$, $-\beta$ and so on will denote the corresponding negations.

** For convenience, we use the standard shorthand "≤" to indicate "< or =". Thus, $n < m$ implies $s_\vee(n) \leq m$.

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

$p_0 = s_\alpha(n)$;

$p_i = s_\vee(p_{i-1})$ for $i = 1, ..., r$ (if $r > 0$);

$s_\vee(p_r) = s_\vee(n)$.

The skeletons of action nodes are empty. The properties of the skeletons of decision nodes will be spelt out in the following lemmas and proposition.

**Lemma 2.1**

Given a decision node $n$, if one of its skeletons $\mathbf{q}_\alpha(n)$ is non-empty, then there exists an elementary succession path $<n, s_{-\alpha}(n), ..., \mathbf{end}>$.

*Proof*:

Assume that no elementary succession path from $n$ passes through $s_{-\alpha}(n)$. Then all elementary succession paths from $n$ pass through $s_\alpha(n)$, and hence $n < s_\alpha(n)$. Therefore, by definition of least common successor, $s_\vee(n) \leq s_\alpha(n)$. But since $s_\alpha(n)$ immediately follows $n$, we must have $s_\vee(n) = s_\alpha(n)$. In other words, $\mathbf{q}_\alpha(n)$ is empty. □

**Lemma 2.2**

Given a decision node $n$, any elementary succession path $<s_\alpha(n), ..., \mathbf{end}>$ contains all the elements of the corresponding skeleton $\mathbf{q}_\alpha(n)$.

*Proof*:

Let $\mathbf{q}_\alpha(n) = <m_0, ..., m_r>$. Then

$m_0 = s_\alpha(n)$;

$m_i = s_\vee(m_{i-1})$ for $i = 1, ..., r$ (if $r > 0$).

Therefore $m_0 \leq ... \leq m_r \leq s_\vee(n) \leq \mathbf{end}$. □

Based on these two lemmas, we can derive Proposition 2.3.

**Proposition 2.3**

Given any decision node $n$, if one of its skeletons $\mathbf{q}_\alpha(n)$ contains $n$, then the opposite skeleton $\mathbf{q}_{-\alpha}(n)$ must be empty.

*Proof*:

Assume the contrary. By Lemma 2.1, there exists an elementary succession path $<n, s_\alpha(n), ..., \mathbf{end}>$. But by Lemma 2.2, $n$ is in $<s_\alpha(n), ..., \mathbf{end}>$. This contradicts the definition of elementary succession paths. □

## 3. MODULES

For any node $n$ in a program flowgraph, we define the *minimal module containing n* (or simply the *module* $M_n$) as a subset of the flowgraph satisfying three 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 $M_n$.

(c) No other node is in $M_n$.

The module $M_{\mathbf{begin}}$ consists of the entire flowgraph minus the **end** node.

In general, we can decide whether a node is inside a given module using the following proposition:

**Proposition 3.1**

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

$m_r = n$;

$\mathbf{q}_{\beta_i}(m_i)$ contains $m_{i-1}$ for $i = 1, ..., r$ (if $r > 0$);

$\mathbf{q}_{\beta_0}(m_0)$ contains $p$.

*Proof*:

Suppose $p$ is in $M_n$. By definition, there exists a decision node $m_0$ in $M_n$ such that one of its skeletons $\mathbf{q}_{\beta_0}(m_0)$ contains $p$. If $m_0 \neq n$, there exists a decision node $m_1$ in $M_n$ such that one of its skeletons $\mathbf{q}_{\beta_1}(m_1)$ contains $m_0$. Since $M_n$ is a subset of the program flowgraph, it contains only a finite number of decision nodes. We can therefore reach $n$ by applying the above procedure a finite number of times. Hence there exists a finite sequence of skeletons $<\mathbf{q}_{\beta_r}(m_r), ..., \mathbf{q}_{\beta_0}(m_0)>$ such that

$m_r = n$;

$\mathbf{q}_{\beta_i}(m_i)$ contains $m_{i-1}$ for $i = 1, ..., r$ (if $r > 0$);

$\mathbf{q}_{\beta_0}(m_0)$ contains $p$.

Let us prove the converse. Suppose there is a finite sequence of skeletons $<\mathbf{q}_{\beta_r}(m_r), ..., \mathbf{q}_{\beta_0}(m_0)>$ satisfying the above conditions. Since $M_n$ contains $m_r$ and $\mathbf{q}_{\beta_r}(m)$ contains $m_{r-1}$, $M_n$ must contain $m_{r-1}$. Proceeding this way for a finite number of steps, we can conclude that $M_n$ contains $p$. □

Given a decision node $n$ and a Boolean value $\alpha$, we define a *branch* $B_\alpha(n)$ as the set of nodes satisfying three conditions:

(a) All the nodes in the skeleton $\mathbf{q}_\alpha(n)$ are in $B_\alpha(n)$.

(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 $B_\alpha(n)$.

(c) No other node is in $B_\alpha(n)$.

Similar to Proposition 3.1, the following is useful in deciding whether or not a node appears in a given branch.

**Proposition 3.2**

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

$m_r = n$;

$\beta_r = \alpha$;

$\mathbf{q}_{\beta_i}(m_i)$ contains $m_{i-1}$ for $i = 1, ..., r$ (if $r > 0$);

$\mathbf{q}_{\beta_0}(m_0)$ contains $p$.

*Proof*:

The proof is similar to that of Proposition 3.1. □

Given a node $m$ inside a module $M_n$, the minimal module $M_m$ which contains $m$ is, of course, a subset of $M_n$. But we are also interested in finding out the condition for which the module $M_n$ coincides with the minimal module $M_m$.

**Lemma 3.3**

(a) If a node $m$ is in the branch $B_\alpha(n)$, then the module $M_m$ is a subset of $B_\alpha(n)$.

(b) If $m$ is in the module $M_n$, then $M_m$ is a subset of $M_n$.

The next proposition and corollary immediately follow:

**Proposition 3.4**

Given a node $m$ in a module $M_n$, if a branch $B_\alpha(m)$ of $m$ contains $n$, then $M_m = M_n$.

**Corollary 3.5**

Given a node $m$ in a module $M_n$, if a skeleton $\mathbf{q}_\alpha(m)$ of $m$ contains $n$, then $M_m = M_n$.

## 4. UNSTRUCTUREDNESS

Intuitively, program unstructuredness may be due to three causes: multiple exits in iterations, exits in the middle of selections, and multiple entries in selections or iterations. In the formal definition of unstructuredness, however, we shall exclude exits in the middle of selections. The reasoning is as follows. Suppose we add an exit $p$ in one of the branches $B_\alpha(n)$ of a selection module $M_n$, such that $s_\beta(p)$ is outside $M_n$. If a succession path from $p$ through $s_\beta(p)$ leads to $n$, then $M_p$ will contain multiple iteration exits $n$ and $p$. If a succession path from $p$ through $s_\beta(p)$ leads to some other node $m$ in $M_n$, then $M_n$ will have multiple entries $n$ and $m$. If no succession path from $p$ through $s_\beta(p)$ leads to any node in $M_n$, then $M_p$ will have multiple entry nodes $p$ and $s_\vee(n)$. Thus, we see that exits in the middle of selections occur only in the presence of multiple iteration exits or multiple entries.

We shall now formally define multiple iteration exits, multiple entries and program unstructuredness. 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 and only if it has more than one iteration exits.

A node $p$ in the module $M_n$ is defined as an entry node of $M_n$ if there exists some node $m$ outside $M_n$ such that $p = s_\alpha(m)$. A module is said to have *multiple entries* if and only if it has more than one entry nodes.

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

## 5. IDENTIFICATION OF MULTIPLE ITERATION EXITS

In this section we shall derive a sufficient and necessary condition for the identification of multiple iteration exits.

**Lemma 5.1**

If a decision node $n$ is in one of the branches $B_\alpha(n)$ and if an elementary succession path $<n, s_\alpha(n), ..., \textbf{end}>$ exists, then there is another decision node $m$ in $B_\alpha(n)$ such that $m$ is an iteration exit of the module $M_n$.

*Proof*:

By Proposition 3.2, since $n$ is in $B_\alpha(n)$, there exists a finite sequence of skeletons $<\mathbf{q}_{\beta_r}(m_r), ..., \mathbf{q}_{\beta_0}(m_0)>$ such that

$m_r = n$;

$\beta_r = \alpha$;

$\mathbf{q}_{\beta_i}(m_i)$ contains $m_{i-1}$ for $i = 1, ..., r$ (if $r > 0$);

$\mathbf{q}_{\beta_0}(m_0)$ contains $n$.

Obviously, the elementary succession path $<n, s_\alpha(n), ..., \textbf{end}>$ contains $m_r$ but not $m_0$. In general, from Lemma 2.2, if the path contains $s_{\beta_i}(m_i)$, it will also contain $m_{i-1}$. Hence there exists some $m_j$ such that the path contains $m_j$ but not the corresponding $s_{\beta_j}(m_j)$. For convenience, we shall denote this $m_j$ by $m$ and denote the corresponding $\beta_j$ by $\gamma$. Clearly $m \neq n$.

By Proposition 3.1, $M_m = M_n$. By Proposition 3.2, $B_\gamma(m)$ contains $m$. We need only prove that $B_{-\gamma}(m)$ does not contain $m$. Assume the contrary. By Lemma 3.3, $B_{-\gamma}(m)$ also contains $n$. By Proposition 3.2, there exists a sequence of skeletons $<\mathbf{q}_{\delta_s}(p_s), ..., \mathbf{q}_{\delta_0}(p_0)>$ such that

$p_s = m;$

$\delta_s = -\gamma$;

$\mathbf{q}_{\delta_k}(p_k)$ contains $p_{k-1}$ for $k = 1, ..., s$ (if $s > 0$);

$\mathbf{q}_{\delta_0}(p_0)$ contains $n$.

Let $p$ be $p_{s-1}$ if $s > 1$ and let it be $n$ otherwise. By Lemma 2.2, since $p$ is in $\mathbf{q}_{-\gamma}(m)$, it lies on the elementary succession path $<n, s_\alpha(n), ..., \textbf{end}>$. If $p = p_{s-1}$, this contradicts our earlier construction that $s_\gamma(m)$ is not in the path. If $p = n$, it contradicts the definition of elementary succession paths. □

**Lemma 5.2**

If a decision node $n$ is in one of the branches $B_\alpha(n)$ and if an elementary succession path $<n, s_\alpha(n), ..., \textbf{end}>$ exists, then the module $M_n$ contain multiple iteration exits.

*Proof*:

By Lemma 5.1, there exists an iteration exit $m$ in $B_\alpha(n)$. By similar arguments, there exists an iteration exit $p$ in $B_{-\alpha}(n)$. Assume that there is only one iteration exit. Then $m = p$, through which all the elementary succession paths will pass. Hence $s_\vee(n) \leq m \neq s_\vee(m) \leq s_\vee(n)$, which is a contradiction. □

The next proposition therefore follows:

**Proposition 5.3**

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

*Proof*:

By definition, $n$ is in one of the branches $B_\alpha(n)$ and not in the opposite branch $B_{-\alpha}(n)$. Hence $\mathbf{q}_\alpha(n)$ cannot be empty. Assume that $\mathbf{q}_{-\alpha}(n)$ is also non-empty. By Lemma 2.1, there exists an elementary succession path $<n, s_\alpha(n), ..., \textbf{end}>$. By Lemma 5.2, therefore, we have at least two iteration exits. □

We can then derive the following theorem:

**Theorem 5.4**

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

*Proof*:

By Proposition 5.3, $n$ is in one of the branches $B_\gamma(n)$ and the opposite branch $B_{-\gamma}(n)$ is empty. By definition of branches, there exists a decision node $m$ in $B_\gamma(n)$ such that a skeleton $\mathbf{q}_\delta(m)$ of $m$ contains $n$.

Assume that $m \neq n$. By Corollary 3.5, $M_m = M_n$. Since $n$ is the only iteration exit, $m$ must be in both $B_\delta(m)$ and $B_{-\delta}(m)$.

By Lemma 2.1, since $B_\delta(m)$ is not empty, there exists an elementary succession path from $m$ through $s_{-\delta}(m)$. By Lemma 5.1, there exists a decision node $p$ in $B_{-\delta}(m)$ such that $p$ is an iteration exit. Since $n$ is the only iteration exit, we must have $p = n$. Hence all elementary succession paths $<m, s_{-\delta}(m), ...,$ **end**$>$ pass through $n$. By the same argument, all elementary succession paths $<m, s_\delta(m), ...,$ **end**$>$ pass through $n$. Thus, $s_\vee(m) \leq n \neq s_\vee(n)$. This contradicts the fact that, since $n$ is in $\mathbf{q}_\delta(m)$, we must have $s_\vee(n) \leq s_\vee(m)$. □

We shall prove that the converse of the theorem is also true, thus obtaining a sufficient and necessary condition for unique iteration exit in a module.

**Theorem 5.5**

Given a module $M_n$, the node $n$ is the only iteration exit if and only if it is in one of the skeletons $\mathbf{q}_\gamma(n)$.

*Proof*:

We need only prove the converse of Theorem 5.4. If $n$ is in $\mathbf{q}_\gamma(n)$, it must be in $B_\gamma(n)$. By Proposition 2.3, $\mathbf{q}_{-\gamma}(n)$ is empty. In other words, $n$ cannot be in $B_{-\gamma}(n)$, and must therefore be an iteration exit. Assume that there is another iteration exit $m$. There are two possibilities.

(a) $m$ is also in $B_\gamma(n)$. Then there exists $p$ ($\neq n$) in $\mathbf{q}_\gamma(n)$ such that $m$ is in $M_p$. Since $n$ is in $\mathbf{q}_\gamma(n)$, $s_\vee(p) \neq s_\vee(n)$. Hence $s_\vee(m) \leq s_\vee(p) \neq s_\vee(n)$. This contradicts the fact that $M_m = M_n$ for iteration exits.

(b) $m$ is in $B_{-\gamma}(n)$. Then $M_m$ is a subset of $B_{-\gamma}(n)$. Since $M_m = M_n$, $n$ must also be in $B_{-\gamma}(n)$. This contradicts the fact that $n$ is an iteration exit. □

We can therefore arrive at a sufficient and necessary condition for the existence of multiple iteration exits.

**Corollary 5.6**

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

## 6. IDENTIFICATION OF MULTIPLE ENTRIES

The identification of multiple entries is more straightforward.

**Theorem 6.1**

A module $M_n$ has multiple entries if and only if there exists two distinct nodes $p_1$ and $p_2$ in $M_n$ such that

$p_1$ is in some skeleton $\mathbf{q}_{\alpha_1}(m_1)$ of a node $m_1$ outside $M_n$;

$p_2$ is in some skeleton $\mathbf{q}_{\alpha_2}(m_2)$ of a node $m_2$ outside $M_n$.

*Proof*:

The proof of the theorem is obvious. □

## 7. CONCLUSION

We have formally studied the properties of skeletons, modules, branches, iteration exits and entry nodes in program flowgraphs. We have proved that two simple conditions are sufficient and necessary for the identification of program unstructuredness. Namely, a module $M_n$ is unstructured if and only 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 exists two distinct nodes $p_1$ and $p_2$ in $M_n$ such that

$p_1$ is in some skeleton $\mathbf{q}_{\alpha_1}(m_1)$ of a node $m_1$ outside $M_n$;

$p_2$ is in some skeleton $\mathbf{q}_{\alpha_2}(m_2)$ of a node $m_2$ 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] 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, pp. 51−61 (1979).

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

[3] B.W. Boehm, *Software Engineering Economics*, Prentice Hall, Englewood Cliffs, NJ (1981).

[4] 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, pp. 70−88 (1985).

[5] J.L. Elshoff and M. Marcotty, “On the use of the cyclomatic number to measure program complexity”, *ACM SIGPLAN Notices* **13** (12): 29−40 (1978).

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

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

[8] H.D. Mills, "The new math of computer programming", *Communications of the ACM* **18** (1): 43−48 (1975).

[9] 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, pp. 220−262 (1982).

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

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

[12] T.H. Tse, "A unified algebraic view of structured systems development models", Doctoral Consortium, 6th International Conference on Information Systems, Indianapolis, IN (1985).

[13] T.H. Tse, "An algebraic formulation for structured systems development tools", *Hong Kong Computer Journal* **2** (4): 44−49 (1986).

[14] T.H. Tse, "Integrating the structured analysis and design models: An initial algebra approach", *Journal of Research and Practice in Information Technology (formerly the Australian Computer Journal)* **18** (3): 121−127 (1986).

[15] T.H. Tse, "Integrating the structured analysis and design models: a category-theoretic approach", *Journal of Research and Practice in Information Technology (formerly the Australian Computer Journal)* **19** (1): 25−31 (1987).

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

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

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

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

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

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