# The Canonical Functional Design
# Based on The Domination-Relationship Among Data

Sukhamay Kundu
*Computer Science Department, Louisiana State University*
*Baton Rouge, LA 70803, USA (email: kundu@bit.csc.lsu.edu)*

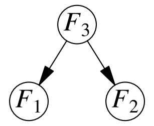
## Abstract

*We study the problem of creating a functional design from a dataflow diagram D. We use the domination-relationship on data-items in D to obtain a canonical function calling-scheme $S(D)$ which is optimal in that it uses the minimum number of global variables for the interface among the functions, while keeping the function-parameters to a minimum. The difficulty in determining a function calling-scheme that is both valid and optimal comes from the fact that the number of valid calling-schemes is exponentially large in the size of D. We also use $S(D)$ to obtain a decomposition of D into larger single-output function-blocks. In [10], we give an algorithm to generate the basic pseudocode for each function, including its interface, for the calling-scheme $S(D)$.*

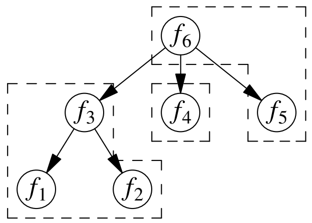
## 1. Introduction

A key step in software design is to create a good functional design for a given algorithm, which may be specified in the form of a high-level pseudocode. This is true irrespective of whether one uses an object-oriented approach or not, and whether the software is large or small. In [13], we read "*Converting a dataflow diagram to a structure-chart (function calling-scheme) is not a mechanical process. It requires designer's insight and creativity. However, there are several 'rules of thumb' $\cdots$*". We show that these rules-of-thumb can be replaced by a precise and theoretically sound method.

A functional design for an algorithm $A$, whose dataflow diagram is $D = D(A)$, consists of two parts: (1) a functional decomposition of $D$, i.e., a partition of $D$ into disjoint subsets $D = D_1 \cup D_2 \cup \cdots \cup D_m$ $(m \geq 2)$ such that the operations in each $D_i$ can be jointly regarded as a mega-function $F_i$, and (2) a suitable calling-scheme among the functions $F_i$. Note that if $S'$ is a calling-scheme for $F_i$'s and we "expand" each $F_i$ in $S'$ in terms of a calling-scheme among its elementary operations (functions) within $D_i$, then we would get a function calling-scheme $S$ for the elementary functions in $D$. Thus, in order for $S'$ to be valid, $S$ itself must be valid and, moreover, the $F_i$'s must form a partition of $S$ into disjoint subtrees as shown in Fig. 1. Thus, we first consider the problem of choosing a proper calling-scheme for the elementary functions in $D$ itself.




(i) A calling-scheme $S'$ among three mega-functions $F_1 = \{f_1, f_2, f_3\}$, $F_2 = \{f_4\}$, and $F_3 = \{f_5, f_6\}$.




(ii) An expanded form of (i) and its associated partition.

Figure 1. A calling-scheme $S$ of elementary functions $f_j$, a partition of $S$ into subtrees, and a calling-scheme $S'$ among the resulting mega-functions $F_i$.

A functional decomposition of $D = D(A)$ gives a grouping of the operations in the algorithm $A$ into disjoint subsets $D_i$; the operations in each $D_i$ form a subalgorithm $A_i$ of $A$, with its own inputs and outputs. In [9], we characterized the subalgorithms $A_i$ as the path-closed subsets of $D$. The function calling-scheme, on the other hand, determines the global variables and the parameters for the functions; they together provide the inputs for each function. (Note that sometimes it is better to use a global variable in order to avoid passing a parameter through a long chain of function-calls for use by a few functions at the end of the calling-chain.) An important objective in a functional design is to keep the interface among the functions, formed by global variables and parameters in the functions, as simple as possible. Two factors which make the functional-design task difficult are: (1) the number of possible function-blocks (i.e., the path-closed subsets) can be exponentially large [9] in the size of $D$, and hence choosing a suitable functional decomposition can be a daunting task, and (2) given a functional decomposition of $D$ into $m \geq 2$ parts, the number of possible valid function calling-schemes is also exponentially large [10] in $m$, and hence choosing a proper calling-scheme is also a non-trivial task.

The well-known notion of program-slices [6] corresponds to a limited class of the path-closed subsets. The path-closedness of a subset $D' \subseteq D$ implies both the logical cohesion of the parts comprising $D'$ and the semantic meaningfulness of $D'$ as a whole [9]. This means the function $f_{D'}$ associated with $D'$ also have logical cohesion and semantic meaningfulness. The input and output data-items for $f_{D'}$ are determined by the connections between $D'$ and its complement $D - D'$.

The traditional method for determining a function calling-scheme [4, 7, 13] is mostly ad-hoc and makes a minimal use of the structure of $D$. We present here a better method based on the domination-relationship in $D$. The key idea is that if a function $f_i$ calls a function $f_j$, then $f_i$ can be regarded as a dominator of $f_j$ in some sense. (If $f_i$ and $f_j$ call each other in a recursive fashion, then neither of them dominates the other.) Since we view the functions to be data-driven, the domination-relationships among the data-items become the natural basis for the functional design. There have been many uses of the low level domination-relationships among variables in a source-code; this includes code optimization in compilers [1, 5], dataflow anomalies [2], and test-path generation in program testing [3, 8]. We use the domination-relationship at a much higher level, namely, for the data-items of a dataflow diagram $D$ of an algorithm $A$, to obtain the canonical function calling-scheme $S(D)$ for $D$, or equivalently, $A$. The calling-scheme $S(D)$ is used, in turn, for determining a functional decomposition of $D$ and the calling-scheme for the resulting mega-functions. We give a complete characterization of the valid function calling-schemes, and show that the number of valid calling-schemes can be exponentially large in the size of $D$.

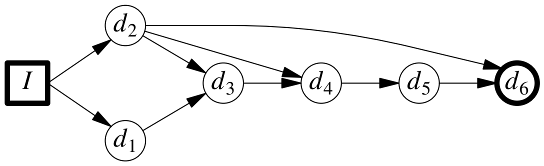
We refer the reader to [10] for the proofs of most of the theorems stated here.

## 2. Dataflow diagram

For our purpose, the exact operations on the data in an algorithm are not as significant as the data-dependency relations which show for each data-item $d_j$ the inputs $d_i$ that are used in computing $d_j$. We therefore use a more abstract form of the usual dataflow diagram, where we do not show the operations and where an arc $(d_i, d_j)$ indicates that $d_i$ is used in computing $d_j$; see Fig. 2. Let $f_j$ denote the operation of computing $d_j$. The operation $f_j$ accounts for all the arcs to node $d_j$; the order in which $f_j$ uses the data-items $\{d_i: (d_i, d_j) \in D\}$ or how they are used by $f_j$ is not of concern here. We assume that $d_i$'s model data-items in the algorithm $A$ at a sufficiently high-level such that $D = D(A)$ is an acyclic digraph [11]. A particular $d_i$ may thus correspond to a whole array, list, etc., or a data file or a single data item as an extreme case; typically, individual items of an array or a list or a file are not modeled as separate $d_i$'s. Likewise, the links in $D$ model operations on an array (list or file) as a whole but not on their individual items. A cycle in $D$ often arises from a low level representation, where a data-item is (or the items of an array or list are) modified iteratively. A simple way to make $D$ acyclic is to replace each strongly connected component by a single node (and analyze each component separately). We can represent a few updated version of a data-item as separate nodes in $D$ and still keep $D$ acyclic. It may happen that the operations $f_i$ and $f_j$ associated with the computation of $d_i \neq d_j$ are the same (but applied to different sets of inputs), but this is not a concern in what follows. We write $\pi(d_i, d_j)$ for a path from $d_i$ to $d_j$ ($\neq d_i$) in $D$. A non-empty subset $D' \subseteq D$ is called *path-closed* [9] if it is connected and for each path $\pi(d_i, d_j)$ connecting two nodes $d_i, d_j \in D'$ we have $\pi(d_i, d_j) \subseteq D'$, i.e., all nodes in $\pi(d_i, d_j)$ belong to $D'$.

**Example 1.** Fig. 2 shows a dataflow diagram for an image processing algorithm considered in [9], where the input $I$ is a black and white image and the output $d_6$ is a simplified output image with some special characteristics. The other $d_i$'s represent intermediate data-items in the algorithm, some of which are lists and the others are arrays of various kinds. The arcs $(d_2, d_4)$ and $(d_3, d_4)$ here indicate that the computation of $d_4$ requires both the inputs $\{d_2, d_3\}$, and likewise for the other nodes. The subset $D' = \{d_2, d_5, d_6\}$ is not path-closed and cannot be considered a function-block or a mega-function [9] because one cannot compute the input $d_4$ to $D'$ before beginning the computations of the data-items $D'$ in the mega-function. This type of problem does not arise when $D'$ is path-closed. The set $D'_1 = \{d_1, d_2, d_3\}$ is path-closed and we may call its associated mega-function "build-the-segment-digraph". The set $D'_2 = \{d_2, d_3, d_4, d_5\}$ is also path-closed and we may call its associated mega-function "build-the-segment-digraph-and-a-longest-path". The set $\{d_1, d_2\}$ is not path-closed since it is not connected; the associated would be mega-function will lack cohesion. There are many other path-closed and non-path-closed subsets of $D$ here. □

We write $in(d_j) = \{d_i: (d_i, d_j) \in D\}$ and $out(d_i) = \{d_j: (d_i, d_j) \in D\}$; let $in^+(d_j) = \{d_i$: there is a path $\pi(d_i, d_j)$ in $D\} \supseteq in(d_j)$ and $out^+(d_i) = \{d_j$: there is a path $\pi(d_i, d_j)$ in $D\} \supseteq out(d_i)$. A node $d_j$ is called an *input node* in $D$ if $in(d_j) = \varnothing$; let $in(D) = \{d_j: in(d_j) = \varnothing\}$. The *output nodes* of $D$ are simply a specially designated subset of nodes, denoted by $out(D)$, which are distinct from the input nodes, i.e., $out(D) \cap in(D) = \varnothing$. In particular, each *sink-node* $d_i$ (i.e., $out(d_i) = \varnothing$) is an output node of $D$. The reader may assume for the initial




$I$ = A black and white input image.
$d_1$ = An array of lists of horizontal segments in $I$; one list for each row of $I$.
$d_2$ = The nodes of an acyclic digraph $G(I)$ related to $I$, with one node for each segment in $d_1$ and the weight of a node being that segment's length.
$d_3$ = The adjacency-lists for the nodes in $G(I)$.
$d_4$ = The matrix $P[i, j]$, which gives the length of a longest path $\pi(n_i, n_j)$ for each node-pair $n_i, n_j \in G(I)$ and the successive nodes on each $\pi(n_i, n_j)$.
$d_5$ = A longest path in $G(I)$.
$d_6$ = The output image.

Figure 2. A dataflow diagram $D$, with the non-input nodes $D_0 = \{d_1, d_2, \cdots, d_6\}$.

reading that the output nodes of $D$ are the same as its sink-nodes. We write $D_0 = D - in(D)$ for the non-input nodes; henceforth, $d_i$ shall denote a non-input node, unless stated otherwise.

A *topological ordering* of $D$ is a linear ordering of the nodes $d_i$ such that each arc in $D$ goes from left to right [1]. We will be concerned here primarily with topological orderings of $D_0$. A topological ordering of $D$ can be obtained by adding the data-items $in(D)$ to the extreme left of a topological ordering of $D_0$ in an arbitrary order. Fig. 2 gives only two topological orderings of $D_0$: $\langle d_1, d_2, d_3, d_4, d_5, d_6\rangle$ and $\langle d_2, d_1, d_3, d_4, d_5, d_6\rangle$, which differ only in the positions of $d_1$ and $d_2$.

## 3. The canonical calling-scheme

We assume for brevity that the software is to be implemented in a C-like language, which does not have the notion of a nested function definition although a similar effect can be achieved by using two or more separately compilable program files. We also assume that $D$ has a unique sink-node $d_N$, where $N = |D_0|$; otherwise, we extend $D$ by adding a dummy sink-node $d_{N+1}$ and the arcs $\{(d_i, d_{N+1})$: $d_i$ is a sink-node in $D\}$. We can think of $d_{N+1}$ as a structure consisting of the components $\{d_i$: $d_i$ is a sink-node in $D\}$; the operation $f_{N+1}$ simply fills in the structure $d_{N+1}$ with its components values. The unique sink-node gives us a tree for the domination-relationship (defined below) instead of a forest.

**Def. 1.** A function *calling-scheme* for $D$ is a rooted, ordered tree $t(D)$ on the nodes $\{f_j$: $d_j \in D_0 = D - in(D)\}$. We sometimes refer to the nodes $f_j$ in $t(D)$ also by $d_j$, and this should not cause any confusion. We write $t_j(D)$ for the subtree at $f_j$ and $parent_t(f_i)$ for the parent of $f_i$ in $t(D)$. The left to right ordering of the children of a node $f_j$ correspond to the order in which those functions are called by $f_j$. □

### 3.1. The domination-relationship

**Def. 2.** Let $D$ be a dataflow diagram with a unique sink-node $d_N$. If every path from $d_i$ to $d_N$ contains $d_j$, we say $d_i$ is *dominated* by $d_j$ and denote this by $d_i \Rightarrow d_j$. (In the literature [5], this is sometimes called post-domination.) □

Clearly, $d_i \Rightarrow d_N$ for each $d_i \neq D_N$. If $d_i \Rightarrow d_j$ and $d_i \Rightarrow d_k$, then exactly one of $d_j \Rightarrow d_k$ and $d_k \Rightarrow d_j$ holds; also, $d_i \Rightarrow d_j$ and $d_j \Rightarrow d_k$ imply $d_i \Rightarrow d_k$, i.e., the domination-relationship is transitive. It follows that the arcs $\{(d_i, d_j)$: $d_i \Rightarrow d_j$ and there is no $d_k$ such that $d_i \Rightarrow d_k$ and $d_k \Rightarrow d_j\}$ form a tree on the nodes of $D$. This is called the *domination-tree* of $D$, denoted by $T(D)$; the parent of $d_i$ in $T(D)$ is called the *immediate dominator* of $d_i$. We write $T_j(D)$ for the subtree of $T(D)$ at $d_j$. It is easy to see that if $d_j = parent_T(d_i)$, then there is a path $\pi(d_i, d_j)$ in $D$; it follows that the nodes in $T_j(D)$ are contained in $in^+(d_j)$. In general, we may have arcs $(d_i, parent_T(d_i)) \notin D$. If $D$ does not have a unique sink-node, then $T(D)$ is a forest and the number of trees in $T(D)$ is at least as large as the number of sink-nodes in $D$. It is heartening to note that if $(d_i, d_j) \in T(D)$, then no $d_{j'} \in in(d_j)$ is dominated by a $d_{i'} \in in(d_i)$; likewise, no $d_i \in in(d_j)$ dominates another $d_{i'} \in in(d_j)$, and no $d_{j'} \in in^+(d_j) - in^+(d_i)$ is dominated by any $d_{i'} \in in^+(d_i)$ since $D$ has no path from $d_{j'}$ to $d_{i'}$.

We assume henceforth that the children of each node in $T(D)$ are ordered from left to right so that the post-order traversal of $T(D)$ gives a topological ordering of $D$. Each arc in $D - T(D)$ is called a *cross-arc* and goes from left to right in $T(D)$; it joins two nodes that are not related by the ancestor-relationship. Each $d_i \in in(D)$ is a terminal node in $T(D)$, and there may be other terminal nodes in $T(D)$. If $(d_i, d_j)$ is the only arc in $D$ from $d_i$, then $(d_i, d_j) \in T(D)$.

**Example 2.** Fig. 3(i) shows the tree $T(D)$ for $D$ in Fig. 2. The arcs in $D \cap T(D)$ are shown here as bold lines and those in $D - T(D)$ as broken lines. □

### 3.2. The calling-scheme $S(D)$

**Def. 3.** The *canonical* calling-scheme $S(D)$ for the functions $\{f_i$: $d_i \in D_0 = D - in(D)\}$ for a dataflow diagram $D$ is given by $T(D) - in(D)$, with the node labels $d_i$ replaced by $f_i$. The tree $S(D)$ is unique, save the ordering of the children of some nodes. We write $S_j(D)$ for the subtree of $S(D)$ at $d_j$. □

**Example 3.** Fig. 3(ii) shows the canonical calling-scheme $S(D)$ corresponding to $T(D)$ in Fig. 3(i). The arcs are directed here from a node to its parent in keeping with the direction of arcs in $T(D)$ and $D$. Fig. 3(iv) shows the basic pseudocode for each $f_j$ based on $S(D)$. We have used $d_i$ to denote both the local variable and the global variable (if any) associated with a data-item $d_i$, and this should not cause any confusion. The call to the root function $f_6$ performs all computations in $D$ via calls to the other $f_i$ and produces the final output $d_6$, using one global variable $d_2$ for the interface among the $f_i$'s. Fig. 3(iii) shows a variation of Fig. 3(ii) with the global variable $d_2$ replaced by a local variable $d_2$ in $f_6$, which is passed as a parameter in each of $\{f_5, f_4, f_3\}$ although $d_2$ is used only in $f_4$ and $f_3$. Fig. 3(v) shows the pseudocode of each $f_i$ for the calling-scheme in Fig. 3(iii). In both Figs. 3(ii)-(iii), we regard the input image $I$ as a global variable (a file, say); alternatively, we can assume it to be a parameter in the root function $f_6$ which writes it to a global variable for use in $f_1$ and $f_2$ or pass it as a parameter to each $f_i$, $i \le 5$. □

The general structure of the pseudocodes in Fig. 3(iv) is shown below, where the set of global variables equals *globalVars* = $\{d_i \in D_0$: there is a cross-arc from $d_i\} \cup in(D)$. If we do not use global variables for $in(D)$, then the parameters in $f_j$ would be $param_j = in^+(d_j) \cap in(D) \supseteq \bigcup\{param_i\text{: } f_i \text{ is a child of } f_j\}$; the parameters of the root function $f_N$ would be $in(D)$ and the parameters in each $f_i$ would be passed by value. The local variables in $f_j$, the global variables used by $f_j$ (and $param_j$, if used) together cover all of $in(d_j) \cap D_0$ (resp., $in(d_j)$). The structure of pseudocodes in Fig. 3(v) is the same, except that there are no global variable now for $d_i \in D_0$ and $param_j$ is slightly more complex.

**Function** $f_j(\,)$
**Local** $\{d_i$: $d_i$ is a child of $d_j$ in $S(D)\}$;
**Global** $in(d_j) \cap globalVars$
1. For (each local variable $d_i$), let $d_i = f_i(\,)$.
2. Compute $d_j$ from $in(d_j)$.
3. **Return** $d_j$.

Suppose $t(D)$ is an arbitrary calling-scheme for $D$ and $(d_i, d_j)$ is a cross-arc. Then, one of the following is true: (1) $d_i$ is a global variable, and (2) $d_i$ is a local variable in the first common ancestor $f_p$ of $N(f_i) = \{f_i\} \cup \{f_k\text{: } d_k \in out(d_i)\}$. In the second case, $d_i$ is passed as a parameter to each function $f_j \ne f_p$ belonging to the $f_k$-$f_p$ path in $t(D)$ for some $f_k \in N(f_i)$; moreover, if $f_p \ne parent_t(f_i)$, then $d_i$ need to be passed by reference to each function $f_j \ne f_p$ on the $f_i$-$f_p$ path in $t(D)$.

**Theorem 1.** If we do not use any $d_i \in D_0 = D - in(D)$ as parameters in the functions $f_j$, then the global variables needed for a calling-scheme $t(D)$ contains those for the canonical calling-scheme $S(D)$.

Proof. If $d_i \in D_0$ and $|out(d_i)| \ge 2$, then it follows from the above observations that $d_i$ will be a global variable for each calling-scheme $t(D)$. On the other hand, if $out(d_i) = \{d_j\}$ and $d_i$ is not a global variable, then $f_j$ must call $f_i$. Both of these properties hold in $S(D)$ and this completes the proof. □

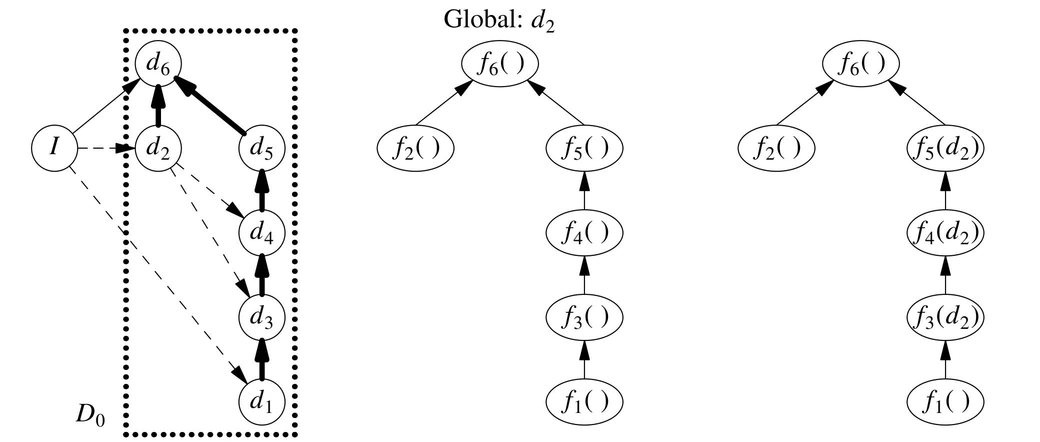
In general, there may be other calling-schemes $t(D) \ne S(D)$ which have the same global variables as those for $S(D)$. For example, if we modify $S(D)$ in Fig. 3(ii) by making $f_2$ a child of either of $\{f_i\text{: } 1 \le i \le 5\}$, then we still have $d_2$ as the only global variable. The pseudocodes in Fig. 3(iv) will have to be modified slightly for the new calling-scheme. Note that if we make $f_3$ call $f_2$, say, then one can use a programming trick to avoid the global variable $d_2$ as follows. We first make $d_2$ a local variable in $f_3$, call $f_2$ in the form as "$d_2 = f_2()$", and then return the value-pair $(d_2, d_3)$ from $f_3$ to $f_4$; similarly, $f_4$ would return the value pair $(d_2, d_4)$ to $f_5$, etc. so that $f_6$ can finally use $d_2$. However, this is basically a disguised form of the use of $d_2$ as a global variable (or as a parameter).

### 3.3. Valid and invalid calling-Schemes

Although many examples of calling-schemes (structure-charts) can be found in the software engineering literature, a precise definition of a valid calling-scheme is missing. This may partly explain the absence of a formal method or algorithm for creating good functional-designs.

**Def. 4.** A function calling-scheme $t(D)$ is *valid* if there is an implementation of each $f_j$ such that $f_j$ calls its child-functions in $t(D)$ in the left to right order, intermixed with the computation of $d_j$ from $in(d_j)$, and the final output $d_N$ is produced by the root-node of $t(D)$. □

**Example 4.** Figs. 4(i)-(iii) show a dataflow diagram $D$ and three potential calling-schemes for it. Fig. 4(ii) is not a valid calling-scheme because $f_2$ must compute $d_2$ before it can call $f_3$ (which uses $d_2$ to compute $d_3$), but in order to compute $d_2$ it needs the value $d_1$ computed by $f_1$ and this means $f_2$ must call $f_3$ (which then calls $f_1$) first according to the calling-scheme. Note that one cannot bypass the problem by introducing parameters in the functions $f_i$, since we can assume that each of $\{d_1, d_2, d_3\}$ is a global variable. If we interchange the positions of $f_1$ and $f_3$ in Fig. 4(ii), we get a valid calling-scheme (cf. Def. 6). Both the calling-schemes in Fig. 4(iii) are valid. For the calling-scheme on the right in Fig. 4(iii), the function $f_2$ first calls $f_1$, then it computes $d_2$ using the value $d_1$ obtained by $f_1$, then it calls $f_3$ to compute $d_3$, and finally $f_2$ returns the value $d_3$. The calling-scheme on the left in Fig. 4(iii) is the canonical calling-schemes $S(D)$. □




(i) $T(D)$ in Fig. 2; the dashed arcs are in $D - T(D)$.

(ii) The canonical calling-scheme $S(D)$.

(iii) Eliminating global variable $d_2$ by using it as a parameter.

---

**Function** $f_1(\,)$:
**Local** $d_1$; **Global** $I$.
1. Compute $d_1$ from $I$.
2. **Return** $d_1$.

**Function** $f_2(\,)$:
**Local** $d_2$; **Global** $I$.
1. Compute $d_2$ from $I$.
2. **Return** $d_2$.

**Function** $f_3(\,)$:
**Local** $d_1$, $d_3$; **Global** $d_2$.
1. $d_1 = f_1(\,)$.
2. Compute $d_3$ from $\{d_1, d_2\}$.
3. **Return** $d_3$.

**Function** $f_4(\,)$:
**Local** $d_3$, $d_4$; **Global** $d_2$.
1. $d_3 = f_3(\,)$.
2. Compute $d_4$ from $\{d_2, d_3\}$.
3. **Return** $d_4$.

**Function** $f_5(\,)$:
**Local** $d_4$, $d_5$.
1. $d_4 = f_4(\,)$.
2. Compute $d_5$ from $d_4$.
3. **Return** $d_5$.

**Function** $f_6(\,)$:
**Local** $d_5$, $d_6$; **Global** $d_2$.
1. $d_2 = f_2(\,)$.
2. $d_5 = f_5(\,)$.
3. Compute $d_6$ from $\{d_2, d_5\}$.
4. **Return** $d_6$.

---

(iv) The basic pseudocode for each $f_i$ in Fig. 3(ii), showing the global variables used in it. Here, the global variables are $d_2$ and $I$.

---

**Function** $f_1(\,)$:
**Local** $d_1$; **Global** $I$.
1. Compute $d_1$ from $I$.
2. **Return** $d_1$.

**Function** $f_2(\,)$:
**Local** $d_2$; **Global** $I$.
1. Compute $d_2$ from $I$.
2. **Return** $d_2$.

**Function** $f_3(d_2)$:
**Local** $d_1$.
1. $d_1 = f_1(\,)$.
2. Compute $d_3$ from $\{d_1, d_2\}$.
3. **Return** $d_3$.

**Function** $f_4(d_2)$:
**Local** $d_3$, $d_4$.
1. $d_3 = f_3(d_2)$.
2. Compute $d_4$ from $\{d_2, d_3\}$.
3. **Return** $d_4$.

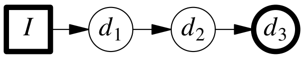
**Function** $f_5(d_2)$:
**Local** $d_4$, $d_5$.
1. $d_4 = f_4(d_2)$.
2. Compute $d_5$ from $d_4$.
3. **Return** $d_5$.

**Function** $f_6(\,)$:
**Local** $d_2$, $d_5$, $d_6$.
1. $d_2 = f_2(\,)$.
2. $d_5 = f_5(d_2)$.
3. Compute $d_6$ from $\{d_2, d_5\}$.
4. **Return** $d_6$.

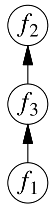
---

(v) The basic pseudocode for each $f_i$ in Fig. 3(iii), without using the global variable $d_2$.

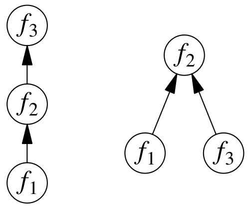
Figure 3. Illustration of the canonical calling-scheme $S(D)$.




(i) A dataflow diagram $D$.




(ii) An invalid calling-scheme for $D$.




(iii) Two valid calling-schemes for $D$.

Figure 4. Examples of valid and invalid calling-schemes.

**Def. 5.** A function calling-scheme tree $t(D)$ is called *regular* if the post-order traversal of $t(D)$ gives a topological order of $D_0$ when we replace each $f_i$ by $d_i$. In particular, the root of $t(D)$ is $f_N$. □

Clearly, the canonical calling-scheme $S(D)$ is regular. The calling-scheme on the right in Fig. 4(iii) is not regular.

**Theorem 2.** A regular calling-scheme is valid.

Proof. Shown below is a pseudocode for each $f_j$ based on a regular $t(D)$. Here, we do not attempt to reduce the number of global variables or the parameters in the functions, although this can be done along the lines given in Section 3.2. Thus, each $d_j \in D_0$ is now taken to be a global variable; the parameters of $f_j$ are given by $param_j = \bigcup\{in^+(d_i) \cap in(D)$: $d_i$ a terminal node in the subtree $t_j(D)\}$. Clearly, $param_j = \bigcup\{param_i$: $parent_t(f_i) = f_j\}$ if $f_j$ is non-terminal and, in particular, $param_N = in(D)$ for the root function $f_N$ in $t(D)$. The step "Compute $d_i$ from $in(d_j)$" in the pseudocode below can be executed properly because, by the regularity of $t(D)$, the value of each $d_i \in in(d_j)$ is known at this point. □

**Function** $f_j(param_j)$
**Local** $d_j$; //not to be confused with global variable $d_j$.
1. For (each child $f_i$ of $f_j$ in $t(D)$), let $d_i = f_i(param_i)$.
2. Compute $d_j$ from $in(d_j)$.
3. **Return** $d_j$.

To see that the number of regular calling-schemes grows exponentially in $N = |D_0|$, let $\tau = \langle d_1, d_2, \cdots, d_N \rangle$ be a topological ordering of $D_0$. For each ordered, rooted tree $t$ with $N$ nodes, assign $f_i$ to the $i$th node of $t$ in the post-order traversal. This gives a set of distinct regular calling-schemes $C_\tau = \{t_\tau$: $t$ an ordered, rooted tree with $N$ nodes$\}$ for $D$; the sets $C_\tau$ are clearly disjoint from each other as we vary $\tau$. Thus, #(regular calling-schemes for $D$) = #(topological orderings of $D_0$)×#(ordered, rooted trees with $N$ nodes) ≥ #(ordered, rooted trees with $N$ nodes) = #(binary trees with $N-1$ nodes) = [2(N-1)]!/[$N$!($N-1$)!]. This proves half of the following theorem; see [10] for the other half.

**Theorem 3.** If $|D_0| \geq 4$, then both the number of valid calling-schemes and the number of invalid calling-schemes are exponentially large in $|D_0|$. □

### 3.4. Characterization of valid calling-Schemes

The following definition generalizes the notion of a regular calling-scheme. The calling-scheme shown on the right side in Fig. 4(iii) is an instance of this generalized form.

**Def. 6.** A function calling-scheme $t(D)$ is $D$-ordered if the following two conditions hold (note that if $f_i$ is a child of $f_j$, then $(d_i, d_j)$ need not be in $D_0$):

(5.1) For each $(d_i, d_j) \in D_0$, $f_j$ is to the right of $f_i$ in $t(D)$ in a post-order traversal of $t(D)$ if neither of $f_i$ and $f_j$ is an ancestor of the other.

(5.2) If $f_j = parent_t(f_i)$ and there is an arc in $D$ joining $f_j$ to a node in $t_i(D)$, then there is no arc in $D$ joining a node in $\bigcup\{t_k(D)$: $f_k = f_i$ or $f_k$ is a child of $f_j$ and is to the right of $f_i\}$ to $f_j$. □

The calling-scheme in Fig. 4(ii) does not satisfy (5.2), but both calling-schemes in Fig. 4(iii) satisfy it. A regular calling-scheme $t(D)$ satisfies (5.1) due to the topological ordering property of the post-order traversal of $t(D)$, and it satisfies (5.2) because no arc in $D$ joins an $f_j$ to a descendent of it. Thus, every regular $t(D)$ is $D$-ordered. If $t(D)$ is $D$-ordered, then we can obtain a topological ordering $Z_t(D)$ of $D_0$ as shown below and it is this property that makes a $D$-ordered calling-scheme valid; $Z_t(D)$ gives the post-order traversal of $t(D)$ if and only if $t(D)$ is regular. For some extreme case, $Z_t(D)$ corresponds to the pre-order traversal of $t(D)$, as in the case of the calling-scheme on the right side of Fig. 5(i).

$Z_t(D) = \langle d_j \rangle$, if $t(D)$ consists of a single node $f_j$.
$Z_t(D) = \langle Z_1, Z_2, \cdots, Z_m, d_j, Z_{m+1}, \cdots, Z_n \rangle$, if $f_j$ is the root node of $t(D)$ and $f_{j_1}, f_{j_2}, \cdots, f_{j_n}$ are its children in the left to right order, $f_{j_{m+1}}$ is the first child, if any ($m \geq 0$), such that there is an arc in $D$ from $f_j$ to a node in the subtree at $f_{j_{m+1}}$, and $Z_k$ is the ordering of the nodes obtained for the subtree at $f_{j_k}$.

Given an implementation of a valid calling-scheme $t(D)$, we get a topological ordering of $D_0$ if we follow the call-sequence and record the order in which the data-item $d_i$ are computed by the calls to various $f_i$. The proof of the following theorem (see [10]) shows that each $D$-ordered calling-scheme $t(D)$ has an implementation in which $d_i$ are computed according to the topological ordering $Z_t(D)$ for $D_0$. The structure of the pseudocode for $f_i$ is shown below.

**Theorem 4.** A calling-scheme $t(D)$ is valid if and only if it is $D$-ordered. □

**Function** $f_j(param_j)$
**Local** $d_j$; //not to be confused with global variable $d_j$.

1. For (the first $m_j$ children $f_{j_k}$ of $f_j$ in $t(D)$, $k = 1, 2, \cdots, m_j$), let $d_{j_k} = f_{j_k}(param_{j_k})$.
2. Compute $d_j$ from $in(d_j)$.
3. For (the other children $f_{j_k}$ of $f_j$ in $t(D)$, $k = m_j + 1, m_j + 2, \cdots, n_j$), let $d_{j_k} = f_{j_k}(param_{j_k})$.
4. **Return** $d_j$. (If $f_j$ is the root node of $t(D)$, then this should be "**Return** $d_N$".)

In [10], we give an algorithm PSEUDOCODES to construct the pseudocodes for each $f_i$ as described in Section 3.2 by starting from a dataflow diagram $D$. The algorithm takes $(N^2 \log N)$ time. The same algorithm, with slight modifications, can be used directly for any regular calling-scheme $t(D)$ as well as for a $D$-ordered calling-scheme, without affecting the complexity.

## 4. Functional decomposition and $S(D)$

As we indicated earlier, a function-block or a mega-function in a dataflow diagram $D$ corresponds to a path-closed subset $D' \subseteq D_0$. Some simple examples of path-closed sets are $out^*(d_i) = out^+(d_i) \cup \{d_i\}$ and $in^*(d_j) - in(D)$, where $in^*(d_j) = in^+(d_j) \cup \{d_j\}$. Also, $out^*(d_i) \cap in^*(d_j)$ is path-closed for $d_j \in out^+(d_i)$, or equivalently, $d_i \in in^+(d_j)$; note that for $d_i = d_j$, $out^*(d_i) \cap in^*(d_j) = \{d_i\}$. With each path-closed subset $D'$, we associate the mega-function $f_{D'}$ whose inputs are $in(D') = \{d_i \notin D' \colon out(d_i) \cap D' \neq \emptyset\}$; this is the set of data-items that are not in $D'$ but needed for computing one or more $d_j \in D'$. The outputs of $f_{D'}$ are $out(D') = [out(D) \cap D'] \cup \{d_i \in D' \colon out(d_i) \cap D - D' \neq \emptyset\}$. For $D' = \{d_i\}$, $f_{D'} = f_i$, with its input $in(d_i)$ and output $\{d_i\}$. The function $f_{D'}$ basically groups the operations $\{f_i \colon d_i \in D'\}$ together. To keep the following discussion simple, assume that $D$ has a single output $d_N = out(D)$.

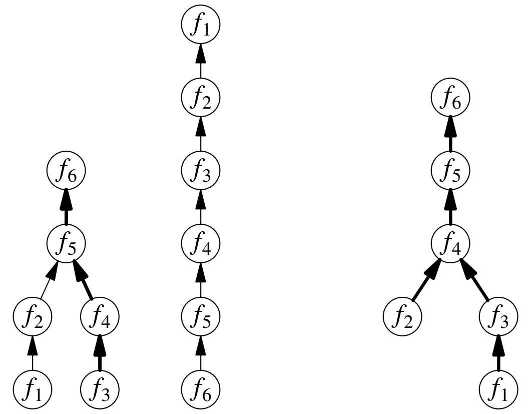
**Def. 7.** A *functional decomposition* $\Pi_m = \{D_1, D_2, \cdots, D_m\}$ of $D$ is a partition of $D_0$ into $m \geq 2$ disjoint path-closed subsets $D_i$. Since each $d_j$ belongs to only one $D_i$, $d_j$ is computed by only one $F_i = f_{D_i}$; it may be used as an input by many $F_k$. □

**Def. 8.** We say the subtree $t_k(D)$ at $d_k$ of a calling-scheme $t(D)$ is *free* if there is no cross-arc $(d_i, d_j)$ where $d_i \notin t_k(D)$ and $d_j \in t_k(D)$. (There may be cross-arcs joining pairs of nodes within $t_k(D)$.) □

Clearly, $t_k(D)$ is free if $d_k$ belongs to the path from the root of $t(D)$ to the leftmost terminal node in $t(D)$. In general, a subtree of a free subtree need not be free. Note that if $t(D)$ is a valid calling-scheme and $D' = t_k(D)$ is free, then $t(D) - t_k(D)$ is valid for the reduced dataflow diagram $D'' = (D - D') \cup out(D')$; the nodes $out(D')$ are treated now as a part of inputs $in(D'')$. If $t(D) = S(D)$, then we also have $t(D) - t_k(D) = S(D) - S_k(D) = S(D'')$, the canonical calling-scheme for $D''$, i.e., for the functions $f_j \notin S_k(D)$.

**Theorem 5.** For each $d_k \in S(D)$, the nodes in the subtree $S_k(D)$ form a path-closed subset of $D_0$ with the unique sink-node $d_k$. The complement $S(D) - S_k(D)$ is also path-closed if $S_k(D)$ is free. □

**Example 5.** For a regular calling-scheme $t(D) \neq S(D)$, $t_k(D)$ may be disconnected, as is the case with $t_2(D) = \{d_1, d_2\} \subseteq D$ the calling-scheme on the leftside of Fig. 5(i). Even if $t_k(D)$ is connected, it may not be path-closed. The second half of Theorem 5 may also be false for a non-regular but valid calling-scheme; for the calling-scheme on right in Fig. 5(i), $t(D) - t_3(D) = \{d_1, d_2\}$ is not path-closed. The regular calling-scheme in Fig. 5(ii) satisfies Theorem 5. □




(i) Two regular calling-schemes $t(D) \neq S(D)$ for $D$ in Fig. 2 which do not satisfy Theorem 5.

(iii) A regular calling-scheme $\neq S(D)$ which satisfies Theorem 5.

Figure 5. Illustration of Theorem 5.

We can use Theorem 5 repeatedly to decompose $S(D)$ from left to right and obtain a functional decomposition $\Pi_m = \langle D_1, D_2, \cdots, D_m \rangle$ for various $m \geq 2$, where each $D_i$ has a unique sink-node and each associated $R_i = D_{i+1} \cup D_{i+2} \cup \cdots \cup D_m$ is also path-closed for $i < m$. In the extreme case, each $D_i$ can be simply $f_i$, with the ordering in $\Pi_m$ being a topological ordering of $D_0$ given by the post-order traversal of $S(D)$. Shown below are the four decompositions obtained from $S(D)$ in Fig. 3(ii) for $m = 3$ once we fix $D_1 = \{d_2\}$. The only alternative choice for $D_1$ according to Theorem 5 is $D_1 = \{d_1\}$

because of the cross-arc $(d_2, d_3) \in D$, and in that case the only choice for $D_2 = \{d_2\}$, giving basically the first decomposition below, with $D_1$ and $D_2$ interchanged.

1. $D_1 = \{d_2\}$, $D_2 = \{d_1\}$, $D_3 = \{d_3, d_4, d_5, d_6\}$
2. $D_1 = \{d_2\}$, $D_2 = \{d_1, d_3\}$, $D_3 = \{d_4, d_5, d_6\}$
3. $D_1 = \{d_2\}$, $D_2 = \{d_1, d_3, d_4\}$, $D_3 = \{d_5, d_6\}$
4. $D_1 = \{d_2\}$, $D_2 = \{d_1, d_3, d_4, d_5\}$, $D_3 = \{d_6\}$

The second decomposition above is the best one for two reasons. First, in terms of the computational load in $D_2$ and $D_3$, this one is more balanced than the others. Second, the dataflow diagram in Fig. 2 is actually a part of a larger one which involves other uses of the longest-paths computed in $d_3$, and hence it is better to choose $D_2 = \{d_1, d_3\}$ with the output $d_3$ so that it can be used for interfacing with other functions not shown in Fig. 2. The mega-functions $F_1$, $F_2$, and $F_3$ associated with the second decomposition have the meanings shown below. The canonical calling-scheme for these functions is given by Fig. 1(i). We can merge $D_1$ and $D_2$ into the path-closed set $D_{12} = \{d_1, d_2, d_3\}$, which together with $D_3 = \{d_4, d_5, d_6\}$ gives the decomposition of $D$ for $m = 2$ which was considered in [9]; here, $D_{12}$ does not have a unique output, but we can make it so by creating a structure with the two components $d_2$ and $d_3$.

$F_1$ = Build segment-nodes, i.e., the nodes of digraph $G(I)$ from the image $I$.

$F_2$ = Build adjacencies among nodes of $G(I)$ and compute longest-paths $\pi_{ij}$ for all node-pairs in $G(I)$.

$F_3$ = Build the new-image from the largest of these longest-paths $\pi_{ij}$.

The canonical calling-scheme for the functions $F_i$ in the general case $\Pi_m = \langle D_1, D_2, \cdots, D_m \rangle$ is also easily obtained from $S(D)$.

## 5. Conclusion

We presented here a sound method for the functional design starting from a dataflow diagram $D$, where we exploit both the local and global properties of $D$ via the domination-relationship of data-items in $D$. The canonical function calling-scheme $S(D)$, which is based on the domination-relationship in $D$, is shown to have several important properties: (1) minimization of the number of global variables that form the interface among the functions while keeping the function parameters to a minimum, and (2) existence of an automated method for generating the basic pseudocode for the functions giving their interfaces (local variables and the ordering of its calls to other functions). We have shown that the canonical calling-scheme can be used also for obtaining a functional decomposition into single-output, larger function-blocks $F_i$, which are semantically meaningful and have logical cohesion. Our future work will address the issue of optimizing a functional decomposition to account for the size (or computational load) of the mega-functions $F_i$ and for its effect on the different top-down and bottom-up software testing methods.

## 6. Reference


1. Aho, A. and Ullman, J.D., *Principles of Compiler Design,* Addison-Wesley Publ. Co., Mass., 1977.
2. Beizer, B. *Software testing techniques,* Van Nostrand Reinhold Comp, New York, 1983.
3. Bertolino, A. and Marré, M., Automatic generation of path covers based on the control-flow analysis of computer programs, *IEEE Trans. on Software Engg., SE*-20(1994), #12, pp. 885-899.
4. DeMarco, T., *Structured analysis and system specification,* Yourdon, Inc, New York, 1978.
5. Ferrante, J., Ottenstein, K.J., and Warren, J.D., The program dependence graph and its use in optimization, *ACM Trans. Prog. Lang. and Systems,* 9(1987), pp. 319-349.
6. Frank, T., A survey of program slicing techniques, *J. of Prog. Lang.,* 3(1995), pp. 121-189.
7. Ghezzi, C., Jazayeri, M., and Mandrioli, D., *Fundamentals of software engineering,* Prentice-Hall, Inc., USA, 1991.
8. G. Rothermel and M.J. Harrold, Selecting Regression Tests for Object-Oriented Software, *Intern. Conf. on Software Maintenance (ICSM94),* pp. 14-25, British Columbia, Canada, Sept. 1994.
9. Kundu, S., The concept of path-closed subsets and its application to software functional design, *Proc. 7th Asia-Pacific Software Engineering Conference, APSEC-2000,* Singapore, Dec. 5-8, 2000.
10. Kundu, S., The canonical functional design based on the domination-relationship among data, Tech. Rep. #2-200, Computer Sc. Dept, Louisiana State University, Baton Rouge, USA.
11. Kundu, S., Normalizing dataflow diagrams for their use as a sound basis for software design, Tech. Rep. #1-2001, Computer Sc. Dept, Louisiana State University, Baton Rouge, USA.
12. Lengauer, T. and Tarjan, R.E., A fast algorithm for finding dominators in a flowgraph, *ACM Trans. Prog. Lang. and Systems.,* 1(1979), pp. 121-141.
13. Sommerville, I., *Software engineering* (5th ed.), Addison-Wesley Publ., England, 1995.
14. Storer, R., *Practical program development using JSP,* Blackwell Scientific Publ., USA, 1987.