NORMALIZED DATAFLOW DIAGRAMS: A SOUND BASIS FOR SOFTWARE DESIGN AND ANALYSIS

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

Abstract

We had shown recently that a carefully prepared dataflow diagram (DFD) can serve as the basis for determining the functional design in an automated (algorithmic) way. The typical informal approach in constructing a DFD may, however, introduce many structural errors making the latter unusuitable for the analysis by those algorithms. We present here a formal definition of a DFD, which imposes a few structural constraints, to eliminate those errors. We show that these constraints do not affect the modeling ability of dataflow diagrams. We give several examples to motivate the structural constraints. A dataflow diagram satisfying these constraints may be called a *normalized* dataflow diagram. We also discuss methods for converting a flowchart to a normalized DFD for reverse engineering.

Keywords: dataflow diagram, normalization, reverse engineering.

1. Introduction

The main purpose of a dataflow diagram (DFD) is to show the major computation steps (processes) in a software and their interfaces, i.e., the inputs and outputs of each process, and the dataflows among the processes in terms of who supplies what inputs to which processes. Creating a proper DFD, however, proves to be a much more difficult task than it might appear (which is amply evident from the many incorrect DFDs that appear in the software engineering literature). The present method of constructing a DFD by hierarchical expansion of processes does not give any structure to a DFD; any set of processes and connection among them repesenting dataflows can be viewed as an expansion of a trivial DFD with just one process. As a result, one cannot associate with a DFD a natural sequence of hierarchical refinements. (For flowcharts, the hierarchical expansion method works successfully because of the structure imposed by the single-input and single-output blocks.) The lack of a formal definition of a DFD has hurt the development of propoer tools and concepts for in-depth analyses of DFDs and their systematic use in the overall software design process and software life-cycle (save some adhoc uses in SADT[4]). Informal descriptions such as "dataflow diagrams illustrate how data flows through a system and how the output is derived from the input through a sequence of functions" or other similar descriptions [4, 7] are not sufficient to formulate precise algorithms for analyzing a DFD. But this situation is beginning to change. In [8-9], we used an abstract form of DFD for obtaining algorithms to determine a functional decomposition of a dataflow diagram and a canonical calling-scheme for those functions. One of the results in [10] shows that if a dataflow diagram Δ satisfies a few simple structural constraints, then we can view Δ in the form of a tree-structure $T(\Delta)$ at a higher level whose refinements gives rise to Δ. The tree $T(\Delta)$ has major implications in both the functional design and the object-oriented design for the underlying software.

Although the low-level dataflows in a program have been used in a variety of ways [1-3, 5, 11-15], this is not so for the high-level notion of DFD. We also do not have a systematic method for creating a high-level DFD from a source program-code, which would be useful as a reverse engineering step for verifying that a program corresponds to a given DFD.

2. Dataflow Diagram

We first present a formal definition of a (normalized) DFD. We give several examples to motivate the restrictions in the definition and show that these restrictions do not limit our ability to model with DFD.

Def. 1. A dataflow diagram consists of a quintuple $\Delta = (I, O, P, G, D)$, where I is a set of external *inputs*, O is a set of external *outputs*, P is a set of *processes*, G is an *acyclic* digraph on the disjoint union $I \cup O \cup P$ whose arcs are contained in $I \times P \cup P \times P \cup P \times O$ (i.e., there is no arc from an input to an output, no arc to an input node, and no arc from an output node), and D is the set of data-items which form the labels on the arcs of G. In addition, the following restrictions hold.

(1) Each process $p_i \in P$ lies on at least one path from an input in I to an output in O and the arcs $(p_i, p_j) \in G$ form a connected graph on P.

(2.1) Each arc of the form $(I_i, p_j) \in I \times P$ has the label I_i. We sometimes draw the arc (I_i, p_j) without showing the node I_i and simply show the label I_i on the arc. Similar remark holds for the arcs of the form $(p_j, O_k) \in P \times O$.

(2.2) An arc (p_i, p_j) may have multiple labels, indicating that each of those outputs of p_i are used by p_j.

(2.3) No two arcs from two different processes p_j, $p_{j'} \in P$ have a common label; in particular, only one process can contribute to an output O_k. A label may appear on several arcs from a process p_j. □

The connectedness assumption in (1) assures us that there is a minimum data dependency among the processes other than perhaps just a coincidental cohesiveness due to one or more common external inputs or outputs. We do not use data-stores as an interface between two processes; they can only serve as external inputs or outputs. The reason for not allowing a cycle in a DFD is explained next.

2.1. Problems with cycles

If there is a cycle $C = \langle p_1, p_2, \cdots, p_n, p_1\rangle$ among the processes in a DFD then no $p_i \in C$ can proceed because every p_i in C depends on its previous process p_{i-1} $(p_0 = p_n)$ for an input. See Fig. 1, where we do not show the dataflow labels on the arcs.

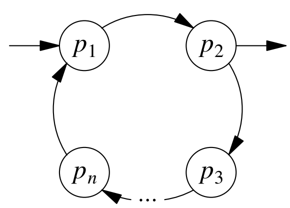
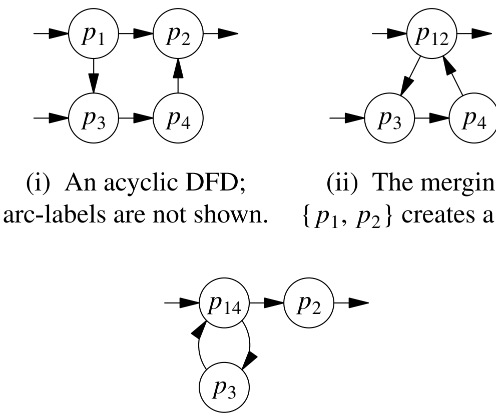


Figure 1. The processes $\{p_i: 1 \le i \le n\}$ in a cycle means a deadlock-like situation.

The simplest way in which a cycle may be ceated in a DFD is by not expanding (detailing) a process sufficiently. In other words, the cycle is created by merging a wrong group of processes into one mega-process to hide details. This is illustrated in Figs. 2(i)-(iii). We can eliminate such cycles by expanding one or more processes in the cycle. In [8], we defined a non-empty subset $P' \subset P$ to be path-closed if P' forms a connected (undirected) subgraph of G and for for each $p_i, p_j \in P'$ and every p_i-p_j path $\pi(p_i, p_j)$ in Δ, we have $\pi(p_i, p_j) \subseteq P'$, i.e., each node in $\pi(p_i, p_j)$ belongs to P'. In Fig. 2, $P' = \{p_1, p_2\}$ is not path-closed because the path $\langle p_1, p_3, p_4, p_2\rangle \not\subseteq P'$; on the other hand, $P' = \{p_1, p_4\}$ is not path-closed because P' is not connected. The connected condition for a path-closed subset P' assures us a minimum cohesion among P', and the other condition $\pi(p_i, p_j) \subseteq P'$ on p_i-p_j paths means that we can regard P' as a whole a mega-process for which we can first compute all its inputs before starting the computations within P'. Theorem 1 below is easily proved. We obviously remove the loop (p', p') created by the merging of P', where p' is the merged node.

Theorem 1. If $P' \subseteq P$ is path-closed, then merging P' into a mega-process does not create a cycle. □



(i) An acyclic DFD; arc-labels are not shown.

(ii) The merging of $\{p_1, p_2\}$ creates a cycle.

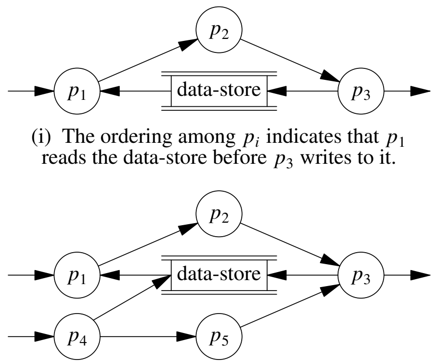
(iii) The merging of $\{p_1, p_4\}$ also creates a cycle.

Figure 2. Creation of a cycle in DFD.

2.2. Cycles involving datastores

Fig. 3(i) shows a DFD which is valid in the conventional sense but not according to our definition. It has a cycle involving a data-store and a chain of processes $C = \langle p_1, p_2, p_3\rangle$ although there are no cycles within P itself. It is implicit here that $p_1 \in C$ reads from the data-store (and accounts for the case when it is empty) before p_3 writes to it. An extreme case of such a cycle arises in an update operation for a data-store involving a chain of just one process, and this occurs frequently in DFDs that appear in the literature. The read/write operations on a data-store by several processes can cause confusion if the DFD represents several walk-thoughs (use-cases). For example, we cannot say whether p_1 in Fig. 3(ii) reads from the data-store before or after p_4 writes to it, although p_4 writes to the data-store before p_3 in the walk-through $\langle p_4, p_5, p_3\rangle$. We can avoid the confusion by introducing a null-dataflow (p_1, p_4), say, to indicate that p_1 reads from the data-store before p_4 writes to it. If several processes read from a data-store and at most one process writes to it, then there is no confusion as long as there is a path from the writing process to each reading process or vice-versa. However, if two unrelated processes (no path between them) write to a data-store, then we have to assume that the order in which they write is irrelevant or we must add a null-dataflow between them. (A cycle involving null-dataflows does not create a deadlock situation.)

In Def. 1, cycles involving a data-store are avoided by the requirement that the inputs I be disjoint from the outputs O. A simple way to break the cycle in Fig. 3(i) is to use two different copies of the data-store, one for each of input and output (see Fig. 4 for an illustration). This can be repeated for each data-store involved in a cycle. The elimination of a cycle is necessary for the analyses given in [8-10].



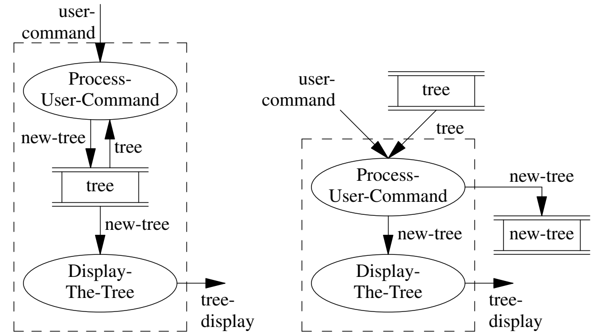
(i) The ordering among p_i indicates that p_1 reads the data-store before p_3 writes to it.

(ii) One cannot say if p_1 reads the data-store before p_4 writes to it.

Figure 3. Cycles involving a data-store and a chain $C = \langle p_1,\, p_2,\, p_3 \rangle$ of processes.

2.3. Data-store is not an internal process-interface

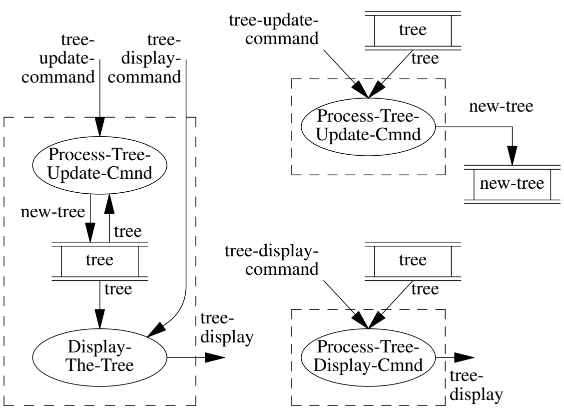
We do not allow a datastore to be an interface between two processes withing a DFD. Fig. 4(i) shows a conventional DFD where one process reads and writes into a data-store and another process only reads from that data-store. Here, a user-command indicates which tree node in the display is to be expanded (to show one or more descendents, if any) or is to be collapsed by removing the node and its descendents from the display. A node may be expanded, say, down to level k from the root or from the node, or to descendents whose subtrees have depth or size at most k, or to descendents which satisfy some other property. The processing of a user-command involves here marking the tree nodes that are to appear in the display, while the actual display is created by the other process. (We assume that if a node is marked, then all nodes on the path from the root to that node are also marked.) The data-store "tree" in Fig. 4(i) acts as the interface between the processes Display-The-Tree and Process-User-Command. Fig. 4(i) does not explicitly indicate that following each user-command the tree-display is refreshed. Fig. 4(ii) corrects this problem. In particular, unlike Fig. 4(i), it represents just one iteration of the cycle of Process-User-Command and Display-The-Tree. Fig. 4(ii) also avoids the cycle in Fig. 4(i) by using two copies of the data-store (the prefix "new" distiguishes the modified data-store from the original one) and keeping the data-store external to the system. Note the use of the dataflow "new-tree" to both the data-store and the process "Display-The-Tree".



(i) A conventional DFD. (ii) A normalized form of (i).

Figure 4. Elimination of a data-store as an interface between two processes within the DFD.

If we have the situation where the user can make several expansion or collapsing operations before the next display (indicated by a separate display-command), then one may represent this in a conventional DFD as shown in Fig. 5(i). This actually overloads the DFD with two distinct use-cases: the update-sequence of the tree and the display-operation; the data-store serves here as a loose connection between the two use-cases. A proper way of modeling this situation is shown in Fig. 5(ii) using two normalized DFDs.



(i) A conventional DFD with two use-cases sharing a common data-store.

(ii) A normalized form of (i) using two DFDs and avoiding the data-store as a direct interface.

Figure 5. Decomposition of an overloaded DFD with two or more use-cases into separate DFDs.

In Fig. 5(ii), the data-store "tree" can represent either a file or an in-memory structure (or both) for the software being modeled. It acts as an *asynchronous* interface between the tree-update operation and the tree-diaplay operation, meaning that each application of one of those operations need not be coordinated with each application of the other.

2.4. Data-store vs. asynchronous process interface

We defined the (normalized) DFD to model processes in a single use-case; these processes are synchronous. Indeed, one can have different DFDs which have one or more common processes; that simply means that the associated sotfware functions will be called in both the use-cases. A software may have one or more its internal datafiles (or some in-memory structure) that are used in processing the inputs in different use-cases. If one of the processes in a use-case must write to one of these internal datafiles and the later is read subsequently (asynchronously) in a different use-case, then we must use a data-store to model this. It may happen that several data items will be written by the first use-case before the second use-case reads a part or all of those new items. If the data written to the data-store is also used in the first use-case itself, then we need to show it as a direct dataflow from the process generating the data instead of a dataflow via the data-store (cf. Fig. 5(i)).

2.5. Decomposition of P for connectedness

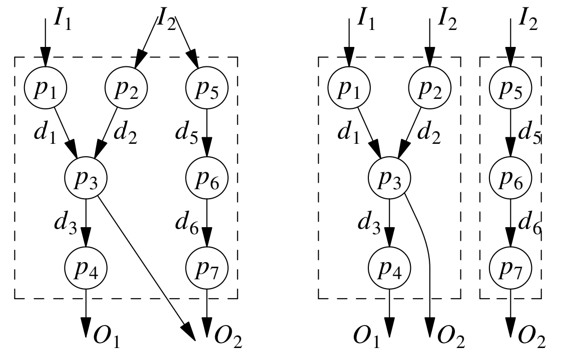
In Fig. 6(i), the processes P satisfy Def. 1 except the connectedness condition (1) and condition (2.3). We can decompose P into two subsets giving two normalized DFDs as shown in Fig. 6(ii), with the processes in each DFD forming a connected set. The duplication of the input I_2 and the output O_2 in the two DFDs is not a problem. Note that if O_2 is a data-store, then in Fig. 6(i) we do not know the order in which p_3 and p_7 may write to O_2, and thus no information is lost in Fig. 6(ii). We can regard the DFDs in Fig. 6(ii) as two separate use-cases that are represented jointly in Fig. 6(i).

3. From Flowchart to DFD

A flowchart contains, in addtion to control flow and detailed computaton steps, information on the dataflows in the program. It should be possible therefore to obtain a DFD from the flowchart as a reverse engineering step, which can be helpful for verifying that a program agrees with the DFD in a design document. Since a DFD does not have the control-flow information, we must absorb each branch-node in the flowchart within a single process. Note that each process shown in a DFD executes once to complete the processing of its inputs I and produce the outputs O. In contrast, a flowchart shows every action that is required for one input or the other. In particular, no input may cause all actions in the flowchart to be executed; a flowchart may also contain paths that are not executed for any input.

3.1. Process for conditional-statement

Figs. 7(i)-(iii) show that we can obtain a normalized DFD from a flowchart in a natural way by regarding



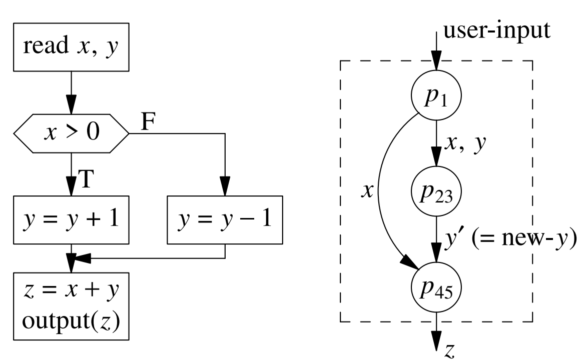
(i) A conventional DFD; G is a disconnected graph on P. (ii) Decomposition of (i) to two valid DFDs.

Figure 6. The decomposition of a disconnected set of processes to create valid DFDs.

each single-input single-output sequential block of the code as a process. In particular, each if-then-else block (or if-then block, with empty else-part) and each loop (including both loop-test and loop-body) is regarded as a process. In Fig. 7(iii), the updated value y' of y due to the if-then-else block forms the dataflow from process p_{23} (for lines 2 and 3 in Fig. 7(i)) to process p_{45} (for lines 4 and 5). Fig. 7(iii) can be simplified slightly by merging p_{23} with p_1 or by merging p_{23} and p_{45}.

1. read x and y;
2. if $(x > 0)$ then $y = y + 1$;
3. else $y = y - 1$;
4. $z = x + y$;
5. output(z);

(i) A code-segment.



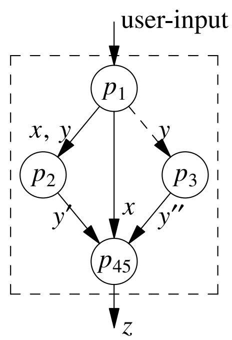
(ii) The flowchart for (i). (iii) A DFD for (ii).

Figure 7. Converting a flowchart into a DFD.

Fig. 8 shows an unsuccessful attempt to obtain a DFD when we split the if-then-else statement in Fig. 7(i) into two statements using a goto (which does not change the flowchart) and we try to build the DFD from the code directly. The problem lies in the multiple return points of process p_2 in Fig. 8(ii) corresponding to line 2 of the code shown in Fig. 8(i); it returns to p_3 when the if-condition fails and to p_{45} otherwise. We indicate this by a dashed dataflow from p_1 to p_3. We also get two inputs y' and y'', both representing the new value of y, to p_4 although only one occurs in any computation. This clearly shows that the advantage of building the DFD from the flowchart. The DFD in Fig, 7(iii) is the result of merging p_2 and p_3 in Fig. 8(ii). Note that merging p_3 and p_{45} in Fig. 8(ii) does not help to eliminate the problem; y and y' would now come to p_{345}.

1. read x and y
2. if $(x > 0)$
   then $\{ y = y + 1;$
   goto stmt 4$\}$
3. $y = y - 1$
4. $z = x + y$
5. output(z)

(i) A code-segment, with flowchart as in Fig. 7(ii).



(ii) A conceivable, but incorrect DFD for (i).

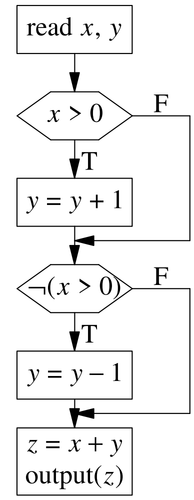
Figure 8. Problems in converting a flowchart with goto's into a DFD.

3.2. Decomposing an if-then-else into two if-thens

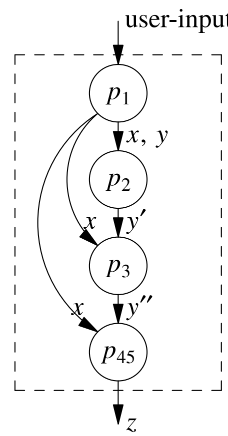
Now consider the effect of splitting the if-then-else statement in Fig. 7(i) into two if-then statements without a goto as shown in Fig. 9(i). (In general, this may require saving the old values of variables used in the if-condition, if some of them are modified in the then-part.) Fig. 9(ii) shows the new flowchart, which is different from Fig. 7(ii). Fig. 9(iii) shows the new DFD. One shortcoming of this DFD is that it fails to show that p_2 and p_3 can be executed in any order. One might be tempted to modify Fig. 9(iii) to the form shown in Fig. 9(iv), but this has the old problem of having two values y' and y'' of y arriving at p_{45} from p_2 and p_3. (Note that interpreting statement-4 as "if $(x > 0)$ then $z = x + y'$ else $z = x + y''$ reintroduces an if-then-else!) The proper solution is to merge p_2 and p_3 in Fig. 9(iv), and we arrive back at Fig. 7(iii). The decomposition of an if-then-else into two if-then statements would work properly if the output variables of the then-part and the else-part are disjoint.

1. read x and y
2. if $(x > 0)$ then $y = y + 1$
3. if $\neg(x > 0)$ then $y = y - 1$
4. $z = x + y$
5. output(z)

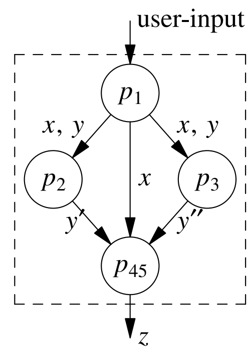
(i) A code-segment.



(ii) The flowchart for (i).



(iii) A correct but undesirable DFD for (i).



(iv) An incorrect DFD after eliminating ordering between p_2 and p_3.

Figure 9. The result of splitting the if-then-else in Fig. 7(i) into two if-then statements.

4. Forming Higher Lelel DFD

Given a large DFD, it is important to be able to form simplified, higher-level DFDs by merging disjoint subsets of processes P'_i at the lower-level to create mega-processes needed for the higher-level DFD. As we saw earlier, each P'_i has to correspond to a path-closed subset of P. If we merge only one P'_i by itself, the path-closed property of P'_i assures us that no cycle will be created and we would get a proper DFD. However, if we merge several P'_i separately, each giving a distinct process node, then we may create a cycle. In Fig. 10, where do not show the dataflow labels, each of $P'_1 = \{p_{11}, p_{12}\}$ and $P'_2 = \{p_{21}, p_{23}, p_{23}, p_{24}\}$ is a path-closed subset, but merging each subset into a distinct process node at the same time gives a cycle between them. If we group p_{21} with P'_1 instead of with P'_2, we also get two path-closed subsets but merging those subsets into single process nodes each again gives a cycle.

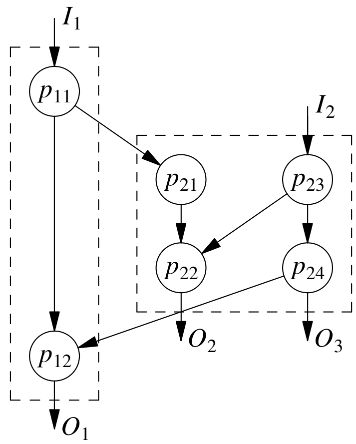


Figure 10. An invalid partition of P into disjoint path-closed subsets for creating a higher-level DFD.

The above example illustrates the difficulty of structuring a large DFD into higher levels. The technique given in [10] decomposes a DFD into disjoint path-closed subsets that gives rise to a higher level DFD having a tree structure. If we apply that algorithm to the DFD in Fig. 10, we get a tree with two nodes corresponding to path-closed subsets $P'_1 = \{p_{11}, p_{21}\}$ and $P'_2 = \{p_{22}, p_{23}, p_{24}, p_{12}\}$.

5. Conclusion

Our main contribution here is a simple formal definition of a dataflow diagram, without compromising its modeling ability. This definition allows us to use the in-depth analyses methods developed previously in [8-10] to support important steps in the software functional design. It can also help in designing the objects or classes in an object-oriented design. We have shown here how to obtain a dataflow diagram from a flowchart as a reverse engineering step.

6. References

1. Beizer, B. *Software testing techniques,* Van Nostrand Reinhold Comp, New York, 1983.
2. Bieman, J.M. and Ott, L.M., Measuring Functional Cohesion, *IEEE Transactions on Software Engineering,* 29(1994)8, pp. 644-657.
3. Bogdan Korel and Janusz Laski, Dynamic program slicing, *Information Processing Letters,* 29(3):155-163, October 1988.
4. DeMarco, T., *Structured analysis and system design, Yourdon Inc., New York, 1979.*
5. Frank, T., A survey of program slicing techniques, *Journal of Programming Languages,* 3(1995), pp. 121-189.
6. Gemino, W. and Wand, Y., Empirical comparison of object-oriented and data-flow models, *Proc. 18th International Conf. Information Systems,* pp. 446-457 1997.
7. Ghezzi, C., Jazayeri, M., and Mandrioli, D., *Fundamentals of software engineering,* Prentice-Hall, Inc., Englewood Cliffs, 1991.
8. Kundu, S., The concept of path-closed subsets and its use in software design, *Asia-Pacific Software Engineering Conference, APSEC-2000, Singapore, pp. 112-119, 2000.*
9. Kundu, S. The canonical functional design based on the domination relationship among data, *Proc. 8th Asia-Pacific Software Engineering Conference, APSEC-2001, Macao, Dec. 4-7, 2001.*
10. Kundu, S., A new analysis of dataflow diagrams for functional and object-oriented design, *The IASTED Intern. Conf. on Software Engineering, SE-2005,* Austria, Feb. 15-17 (to be held).
11. Lakhotia, A., Rule-Based Approach to Computing Module Cohesion, *Proceedings of the 15th International Conference on Software Engineering,* May 17-21, Baltimore, 1993, pp. 35-44.
12. Ott, L.M. and Thuss, J.J., The Relationship between Slices and Module Cohesion, *Proc. 11th International Conference on Software Engineering,* Pittsburgh, May 15-18, 1989.
13. Ott, L.M. and Thuss, J.J., Slice Based Metrics for Estimating Cohesion, *Proceedings of the First International Software Metrics Symposium,* Baltimore, May 21-22, 1993, pp. 71-81.
14. Stevens, W.P., Myers, G.J., and Constantine, L.L., Structured design, *IBM Systems Journal,* 13(1974), pp. 115-139.
15. Weiser, M., Program slicing, *IEEE Trans. Software Engineering,* SE-10, 4(1984), pp. 352-357.