# THE CONCEPT OF PATH-CLOSED SUBSETS AND ITS USE IN SOFTWARE FUNCTIONAL DESIGN

*Sukhamay Kundu*

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

**ABSTRACT.** A key step in software design is the conversion of an algorithm-pseudocode, which emphasizes the solution method and the data-structures at the conceptual level, to a program-pseudocode which gives the overall structure of the functions and global data for implementation. Based on the data-flow of an algorithm, we define the concept of path-closed subsets of the data-items, and an equivalence relation and a partial order on the data-items. We then develop three design rules for creating a functional design from the algorithm-pseudocode. We illustrate our method using an algorithm with complex data-flows and data-structures.

**Keywords:** path-closed subsets, data-flow, functional decomposition.

## 1. INTRODUCTION

A software (program-code) is an embodiment of two important types of knowledge, among other things: (1) A solution algorithm for some problem, which emphasizes the solution method and the data-structures used in it at a conceptual level, (2) A software design for that algorithm, giving the overall structure (functional decomposition, data-organization, etc) of the program. The structure of a software may also be influenced by other phases of its life-cycle, such as testing; for example, a software may contain additional internal data and output-functions to help the debugging and testing operations, which are as such unrelated to the problem being solved. Our focus here is on the software design step, and our method can be considered orthogonal to both Jackson's JSP method [19] and Yourdon's SD/SA method [9].

An algorithm can be implemented by many different programs having different structures and complexity. An important goal in software design is to keep the complexity of information flow to a minimum level. This is a sufficiently complex task in itself that justifies the separation of software design task from the algorithm design. See Fig. 1. Sometimes these two tasks are considered as one and the same, as in many business application-softwares, where the underlying algorithms are not too complex and the software structure closely resembles the algorithm structure. In general, these two structures may be significantly different. For example, depending on the complexity of the algorithm, one may use several functions to implement one step of the algorithm or one may combine several steps into a single function in order to reduce global variables and data-flow among those steps. Likewise, one may decompose one or more high-level data-items in the algorithm into smaller pieces and regroup them differently based on their usage pattern to simplify the data-flow.

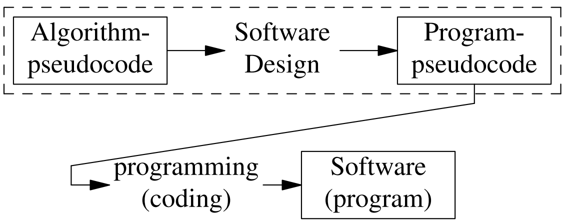



Figure 1. Software design: converting an algorithm-pseudocode to a program-pseudocode.

Shown below are two key tasks (a)-(b) in designing a software for a given algorithm. A large part of these tasks can be automated once we formulate suitable criteria for the grouping, and develop the necessary methods to create an optimal grouping. The present work is a step forward in this direction based on the three concepts introduced here: an equivalence relation, a partial order, and the notion of path-closed subsets of data-items. These concepts make use of both the local and the global properties of the algorithm's data-flow. We show how these concepts can be used systematically to create a functional design starting from the data-flow diagram of a complex algorithm.

(a) Grouping of data-items in the algorithm to simplify the data-flow between functions.

(b) Grouping of operations in the algorithm to form functions, which helps to limit the scope (visibility) of various data-items and hence reduce the information-flow complexity.

There are many good reasons for using the data-flow of an algorithm as the basis for its software design. Consider two programs $P_1$ and $P_2$ which have a similar structure and hence a similar data-flow pattern, but which differ in some key operations. For example, $P_1$ may compute the sum (or sum of squares) of an array or a list of numbers and $P_2$ may compute the maximum (or minimum) of them. It is not surprising that the algorithms associated with $P_1$ and $P_2$ have a similar (high level) data-flow pattern as well. Conversely, if two algorithms $A_1$ and $A_2$ have a similar data-flow pattern but they differ in the nature of their data-items and the operations on them, then the functional designs for $A_1$ and $A_2$ will be similar as well, although the individual functions themselves can be quite different in the two cases.

One of the key concepts developed here is that of a *path-closed subset* of a data-flow diagram. We use this concept in identifying potential subfunctions in an algorithm. A path-closed subset is a high-level generalization of the low-level notion of a program-slice, which is a subset of a program's statements. Program-slices have been used extensibly in program-understanding and debugging, definition-use patterns of variables in a program, program-complexity measures, and program transformations [2, 3, 4, 10, 11, 14, 16, 17, 21]. Since software design has a strong impact on each of these, it is not surprising that path-closed subsets have an important role in the software design for an algorithm.

Data-flow diagrams are often described in Software Engineering literature [5, 9, 13] as an informal aid in developing an algorithm, where a "bubble" corresponding to an operation in the algorithm is refined by adding new data-items and sub-operations. Likewise, a data-item may be decomposed into smaller pieces, with associated changes in the operations involving that data-item. This approach works well for the simplest kind of algorithms as encountered in many business applications. Serious algorithm design for complex problems, such as the algorithm H-CONVEX used below for illustrations, hardly ever relies on such an approach. A data-flow diagram is nevertheless very useful as a descriptive model of an algorithm that is already developed, and hence it can also benefit the software design.

We omit the proofs of various Theorems and Corollaries below for want of space.

## 2. DATA-DIAGRAM

For our purpose, the specific operations used in an algorithm are not of as much concern as are the inputs and outputs of those operations and the data-flow pattern. This suggests the following definition.

**Def. 1.** Given an algorithm $A$, its *data-diagram* $D = D(A)$ is defined to be the digraph where each node $d_i$ corresponds to a high-level data item in $A$ and there is a link $(d_i, d_j)$ if some operation in $A$ uses $d_i$ as one of its inputs and computes $d_j$ as one of its outputs.

A node $d_i$ in $D(A)$ may correspond to a whole data structure (array, list, or tree) or a data file or a single data item as an extreme case. We typically do not model an individual item in an array or list as a separate node $d_i$; likewise, the links in $D(A)$ model operations on an array or a list or a file as a whole and not on their individual items. A single step in the algorithm may give rise to several links $(d_i, d_j)$ to a node $d_j$ and to several such sets of links to different $d_j$'s, depending on the complexity of that step. We assume that the steps in the pseudocode for $A$ clearly specify which inputs $d_i$ are used for computing each output $d_j$ in that step. We assume throughout this paper that the data-items are modeled at a sufficiently high level so that $D(A)$ is acyclic. A cycle in $D(A)$ often arises from a low level representation, where a data-item (or items of an array or list) is modified iteratively. A simple way to make $D(A)$ acyclic is by replacing each strongly connected component of $D(A)$ by a single node (and analyze it separately, which may give rise to a set of recursive functions). We can represent at updated version of a data-item such as a file as a separate node in $D(A)$ and still maintain $D(A)$ to be acyclic, if that data-item is updated a fixed constant number of times. Note that a given link $(d_i, d_j) \in D(A)$ can correspond to only one step in $A$.

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 $\overline{in}(d_j) = \{d_i$: there is a path from $d_i$ to $d_j$ with one or more links$\} \supseteq in(d_j)$; since $D(A)$ is acyclic, $d_j \notin \overline{in}(d_j)$. The notion of $\overline{out}(d_i)$ is defined in a similar way, and we have $\overline{out}(d_i) \supseteq out(d_i)$ and $d_i \notin \overline{out}(d_i)$. The data-items $\overline{in}(d_j)$ directly or indirectly influence $d_j$ and the data-items $\overline{out}(d_i)$ are directly or indirectly influenced by $d_i$. We write $span(d_i) = \{d_k$: $d_k$ belongs to a path $\pi(d_i, d_j)$ for some $d_j \in \mathrm{out}(d_i)\}$. Clearly, $out(d_i) \subseteq \mathrm{span}(d_i) \subseteq \overline{out}(d_i)$. A closely related concept is $scope(d_i)$ = the complement of $\overline{in}(d_i) \cup \{d_i\}$ = the set of data-items where the value of $d_i$ is available for use; $\mathrm{scope}(d_i) \supseteq \overline{out}(d_i) \supseteq \mathrm{span}(d_i)$. Note that if $(d_i, d_j)$ and $(d_j, d_k) \in D$, then $(d_i, d_k)$ may or may not be in $D$. For our purpose, the arcs from $in(d_j)$ to $d_j$ are viewed as an "and" combination in that the computation of $d_j$ uses them together. We write $f_{d_i}$ for the operation of computing $d_i$ from $in(d_i)$.

We write $in(D) = \{d_j: in(d_j) = \varnothing\}$; each $d_i \in in(D)$ is called an *input* (source) node of $D$. The *output* nodes of $D$, denoted by $out(D)$, is simply a set of distinguished nodes, which correspond to the outputs of the algorithm $A$, and includes the *sink* nodes $sink(D) = \{d_i: out(d_i) = \varnothing\}$. In particular, some $d_i \in out(D)$ may be used for computations leading to other $d_j \in out(D)$. This happens, for example, if the algorithm outputs both the average and the variance of a set of numbers, where the average is used in computing the variance.

**Example 1.** Fig. 2 shows the data-diagram for a complex algorithm H-CONVEX, whose pseudocode is shown in Appendix. We use the array-form "left[$k$], $1 \le k \le$ numSegments" to represent "left($s_k$) for all segments $s_k$" for notational convenience; similarly, for right($s_k$), size($s_k$), row($s_k$), adj($s_k$), longestPathLength($s_k$), and parentInPath($s_k$). The value "numSegments" is an implicit output of step (1) of H-CONVEX and is not shown here as a separate $d_j$ to keep the diagram simple; it basically serves as a descriptor of some of the arrays. For the same reason, we do not show $M$ and $N$ for the array image[$i$, $j$] as separate data-items. The details in steps 2(a)-(b) of H-CONVEX, which involve operations on the individual items in the array adj[$k$] and other arrays, are not modeled in Fig. 2. Note that step (3) of H-CONVEX clearly states that both $d_4$ and $d_6$ are used for computing each of $d_7$ and $d_8$. If we simply stated "find longestPathLengths", then the presence of the nodes $d_7$ and $d_8$ and the links to them in Fig. 2 could not be easily determined without a detailed knowledge of a longest-path algorithm. On the other hand, if steps (3)-(4) in H-CONVEX were combined and stated as "find a longest path $\pi$ using adj[$k$]'s and size[$k$]'s", then the nodes $\{d_7, d_8\}$ would be eliminated from the data-diagram and we would have the direct links $\{(d_4, d_9), (d_6, d_9)\}$. Our software design method described in Sections 3 and 4 can be applied to the resulting simpler data-diagram as well. The step (5) of H-CONVEX states that $\{d_2, d_3, d_5, d_9\}$ are used in computing the output image. □

**Def. 2.** We say $d_i$ is *input-equivalent* to $d_j$, denoted by $d_i \approx_I d_j$, if $in(d_i) = in(d_j)$. The notion of *output-equivalence* $d_i \approx_O d_j$ is defined in a similar way. Finally, we say $d_i$ is *equivalent* to $d_j$, denoted by $d_i \approx d_j$, if we have both $d_i \approx_I d_j$ and $d_i \approx_O d_j$. □

Clearly, each of "$\approx$", "$\approx_I$", and "$\approx_O$" is an equivalence relation. If $d_i \approx_I d_j$ and $d_i \in in(D)$, then $d_j \in in(D)$. The same is true if we replace "$\approx_I$" by "$\approx$". However, if $d_i \approx_O d_j$ and $d_i \in out(D)$, then we may or may not have $d_j \in out(D)$. If $d_i \approx d_j$, then there is no path from $d_i$ to $d_j$ because $D$ is acyclic. Note that "$\approx$" depends only on the local properties of $D$.

**Def. 3.** We say $d_i$ *input-dominates* $d_j$, denoted by $d_i >_I d_j$, if $in(d_i) \supset in(d_j) \neq \emptyset$. The condition $in(d_j) \neq \emptyset$ avoids the trivial orderings $d_i >_I d_j$, for $d_i \notin in(D)$ and $d_j \in in(D)$. The notion of *output-dominance* $d_i >_O d_j$ is defined in a similar way by $out(d_i) \supset out(d_j) \neq \emptyset$. Finally, $d_i$ *dominates* $d_j$, denoted by $d_i > d_j$, if $d_i >_I d_j$ and $d_i >_O d_j$. If we use $\overline{in}(d_i)$ and $\overline{in}(d_j)$ in place of $in(d_i)$ and $in(d_j)$, then the resulting partial order is denoted by $\gg_I$; since $d_j \in \overline{in}(d_i)$ implies $\overline{in}(d_i) \supset \overline{in}(d_j)$, the only cases that are of interest to us are when $d_j \notin \overline{in}(d_i)$. The partial orders "$\gg_O$" and "$\gg$" are defined in a similar way. □

**Example 2.** In Fig. 2, we have $d_2 \approx d_3$ and $d_7 \approx d_8$. On the other hand, $\{d_1, d_2, d_3, d_4, d_5\}$ are input-equivalent and $\{d_4, d_6\}$ are output-equivalent as are $\{d_5, d_9\}$; there are no other input-equivalence or output-equivalence except for those implied by "$\approx$". No two data-items are ordered here by "$>_I$", and hence by "$>$". We have each of $\{d_2, d_3\}$ is $>_O$ each of $\{d_1, d_5, d_9\}$; we also have $d_2, d_3 \gg_O d_1 \gg_O d_4 \gg_O d_5$, where we can replace $d_4$ by $d_6$ and/or replace $d_5$ by $d_9$. □

The following theorem, which is easily proved, shows that the partial orderings in Def. 3 can be extended to equivalence classes of "$\approx$". (This is because "$\approx$" could be defined in terms of "$>_I$" and "$>_O$".)

**Theorem 1.** The equivalence relation "$\approx$" is consistent with the various partial orderings in Def. 3. In particular, $d_i >_O d_j$ and $d_j \approx d_k$ implies $d_i >_O d_k$, and also $d_i \gg_O d_j$ and $d_j \approx d_k$ implies $d_i \gg_O d_k$. Likewise, $d_i \approx d_k$ implies $d_k >_O d_j$ and $d_k \gg_O d_j$. □

## 3. TWO ELEMENTARY DESIGN RULES

Our first two design rules DR-1 and DR-2 are based on "$\approx$" and "$\gg_O$". They help to simplify $D$ by merging two or more $d_i$'s into a single mega-data item, in preparation for the functional design described in Section 4. The merging of $d_i$ and $d_j$ into the mega-data item $d_{ij}$ means the operations $f_{d_i}$ and $f_{d_j}$ for computing them are also combined into a mega-operation $f_{d_{ij}}$ which computes both $d_i$ and $d_j$. We typically apply DR-2 only after the applications of DR-1, although the final result does not depend on the order in which they are applied. We do not have a design rule to merge two disjoint isomorphic subdiagrams $D_1$ and $D_2$ of size > 1 because $f_{d_j}$ for $d_j \in D_1$ may be a different operation than $f_{d_{j'}}$ for the corresponding $d_{j'} \in D_2$. Even if we had $f_{d_j} = f_{d_{j'}}$ for every $d_j \in D_1$ and its corresponding $d_{j'} \in D_2$, it may happen (see Section 4) that we can form a function $f_{D_1}$ corresponding to $D_1$ but we cannot form a function $f_{D_2}$ corresponding to $D_2$ because of the differences in the links connecting $D_2$ and $D - D_2$ and those connecting $D_1$ and $D - D_1$. However, if we could form both $f_{D_1}$ and $f_{D_2}$, then they would correspond to two applications of the same function to two disjoint set of data-items.)

**Rule DR-1:** If $d_i \approx d_j$, then merge them into a single data-item $d_{ij}$, except when one of them is in out($D$) and the other is not.

**Rule DR-2:** If $d_i \gg_O d_j$ and $in(d_i) = in(d_j)$, then merge them into a single data-item $d_{ij}$, except when one of them is in out($D$) and the other is not.

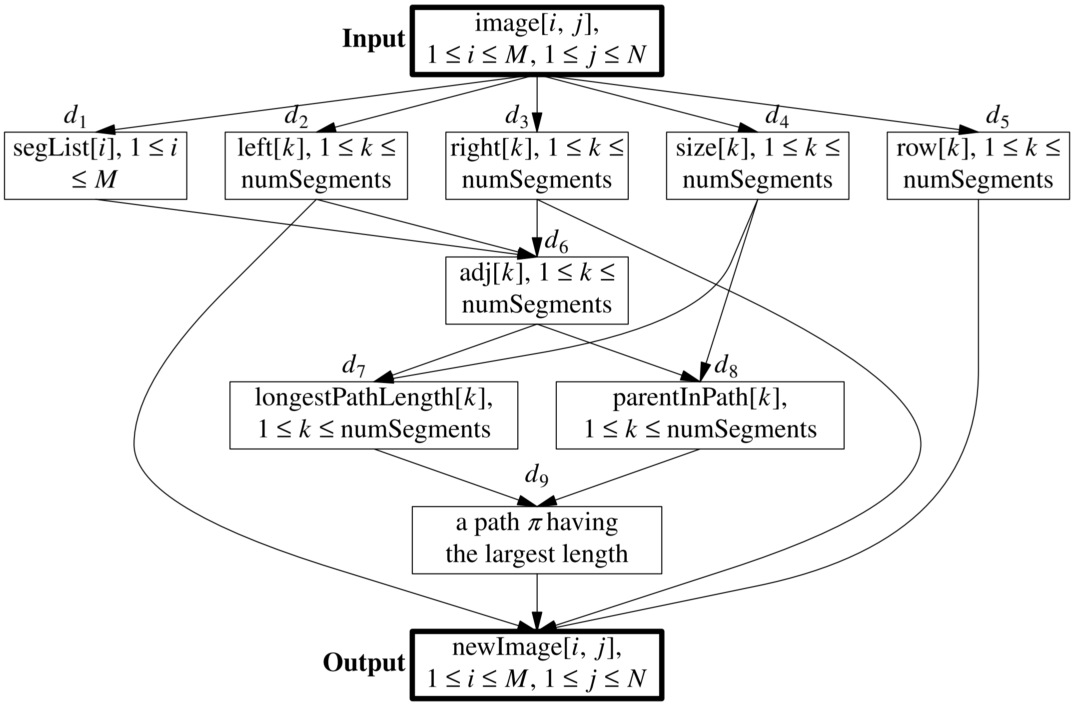
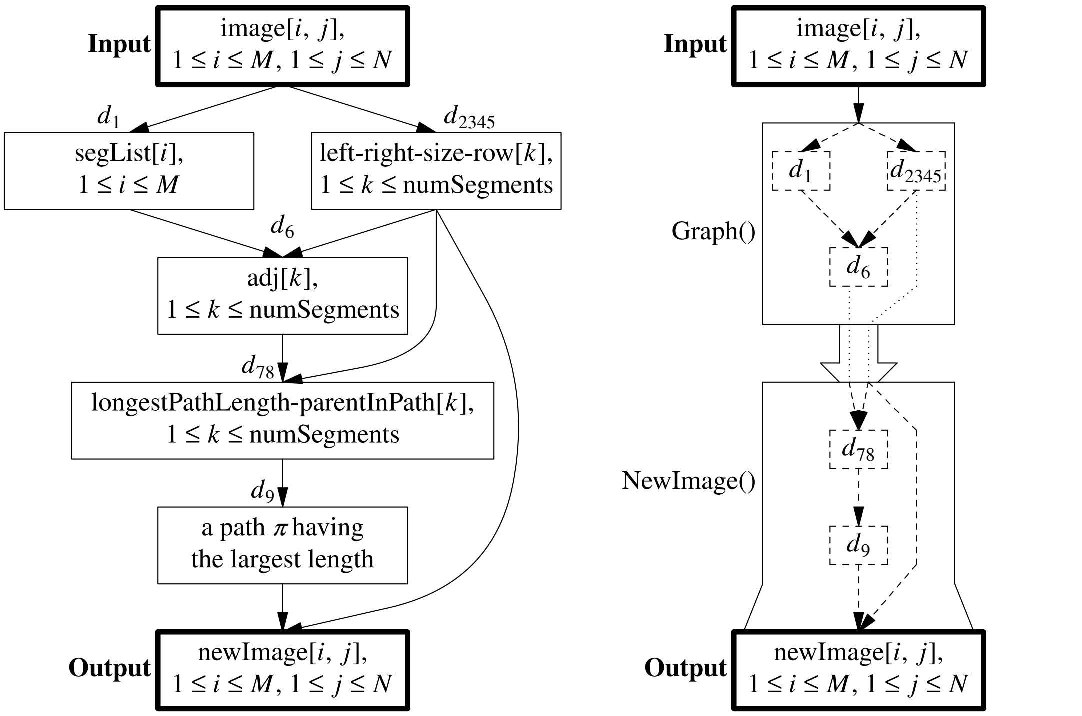



Figure 2. The data-diagram for algorithm H-CONVEX in Appendix-B.




(i) A reduced data-diagram obtained from Fig. 2 using DR-1 and DR-2.

(ii) Additional data-grouping $\{d_1, d_{2345}, d_6\}$ and $\{d_{78}, d_9, \text{output}\}$ using DR-3.

Figure 3. Illustration of design rules DR-1, DR-2, and DR-3.

The new data-diagram obtained by an application of DR-1 has $in(d_{ij}) = in(d_i) = in(d_j)$ and $out(d_{ij}) = out(d_i) = out(d_j)$; an application of DR-2 will result in $in(d_{ij}) = in(d_i) = in(d_j)$, $out(d_{ij}) = out(d_i) \cup out(d_j)$, and $\overline{out}(d_{ij}) = \overline{out}(d_i)$. The condition "$in(d_i) = in(d_j)$" in DR-1 (implied by "≈") and DR-2 assures us that the components $f_{d_i}$ and $f_{d_j}$ in the mega-operation $f_{d_{ij}}$ have communicational-cohesion [14, 18]. Also, without this condition in DR-2, one may create a loop by an application of DR-2; for example, if $D$ consists of a single path connecting an input node to an output node, then without the condition "$in(d_i) = in(d_j)$" in DR-2 we can merge all non-output nodes into a single node with a loop at that node. The rule DR-2 is somewhat weak in that it may allow too much merging (see Example 3). We can strengthen DR-2 by replacing "$\gg_O$" with "$>_O$".

Neither of DR-1 and DR-2 allows us to merge an input node with a non-input node, and likewise an output node with a non-output node. This is justified by the fact that we want to keep the inputs and outputs free of any extraneous data. To implement $d_{ij}$ in a program-code, we can use a structure (record) with two parts, one for each of $d_i$ and $d_j$. If $d_i$ and $d_j$ are arrays of the same size and dimension, then $d_{ij}$ can also be an array of the same dimension, where each item of the array for $d_{ij}$ is made of the corresponding components of the arrays for $d_i$ and $d_j$. Similarly for the case where $d_i$ and $d_j$ are lists of the same size.

**Example 3.** We obtain $d_{2345}$ in Fig. 3(i) from Fig. 2 by first forming $d_{23}$ from $\{d_2, d_3\}$ by DR-1, and then combining $d_{23}$ with $d_4$ and $d_5$ by two applications of DR-2. Another application of DR-1 to $\{d_7, d_8\}$ gives $d_{78}$. We do not merge $d_{2345}$ and $d_1$ by DR-2 partly because they are of non-matching type; $d_{2345}$ is an array and $d_1$ consists of several lists of possibly varying sizes (moreover, $M$ may be different from numSegments). There are other reasons for not merging $d_{2345}$ and $d_1$, which will become clear in Section 4. If we do not explicitly store size[$k$] = right[$k$] − left[$k$] + 1 in algorithm H-CONVEX, then $d_4$ will be removed from Fig. 2 and the links $\{(d_4, d_7), (d_4, d_8)\}$ will be replaced by $\{(d_2, d_7), (d_3, d_7), (d_2, d_8), (d_3, d_8)\}$. This will not, however, change the equivalence $d_2 \approx d_3$ and Fig. 3(i) will remain the same except that $d_{2345}$ will now become $d_{235}$ and we will write left-right-row[$k$] for it. □

The application of DR-2 to form $d_{234}$ from $\{d_{23}, d_4\}$ in Fig. 3(i) can be thought of as momentarily replacing the links $\{(d_4, d_7), (d_4, d_8)\}$ by $\{(d_4, d_6), (d_4, \text{output})\}$ and then applying DR-1. A similar remark holds, in general, where the links from $d_j$ to $out(d_j)$ are momentarily replaced by the links $\{(d_j, d_k): d_k \in out(d_i)\}$, creating an artificial indirect-dependency of $out(d_j)$ on $d_j$ via $out(d_i)$ and making $d_j \approx d_i$ momentarily to allow the combining of $d_i$ and $d_j$ by DR-1. This type of merging two or more non-equivalent data-items is commonly used by programmers, and this partly explains why understanding a program (or a program-pseudocode) is often more difficult than understanding its algorithm-pseudocode. Table 1 shows the effect on the program-pseudocode due to merging of two data-items by DR-1 or DR-2. The effect on the program-code will be similar and straightforward.

TABLE 1. The effect of merging $d_i$ and $d_j$ by DR-1 or DR-2 on a program-pseudocode.

```
Before merging:
    Compute d_i from in(d_i) (= in(d_j));
    Compute d_j from in(d_j);
After merging:
    Compute the parts d_i and d_j of d_ij
    from in(d_i) (= in(d_j));
```

We note in passing that if we can form $d_{ijmn}$ by first forming $d_{ij}$ and $d_{mn}$ via DR-1, and then combining them to form $d_{ijmn}$ via DR-2, then we can alternately form $d_{im}$ and $d_{jn}$ first via DR-2 and then combine them to form $d_{imjn}$ via DR-1. In either case, we can consider $d_{ijmn} = d_{imjn}$ and all the fields in them to be at the same level instead of forming a nested structure of various fields and subfields. A nested structure within a record basically helps to access certain subset of the fields simultaneously, say, for an assignment operation, but it does not otherwise significantly affect the computations in a program-code.

If we design a software based on Fig. 3(i) using a separate function $f_{d_i}$ for computing each $d_i$ from the inputs $in(d_i)$, then we need 5 major global variables, one for each $d_i$ other than the input and output nodes, save the variables for $M$, $N$, and numSegments. This is much better compared to 9 major global variables needed if we use Fig. 2. On the other hand, if we use Fig. 3(i) to create one function H-CONVEX for all computations, then we need 5 key local variables playing the role of the five global variables mentioned above, and this does not reduce the information complexity. The design rule DR-3 given in the next Section will give us a functional design, which reduces the number of major global variables from 5 to 1, by localizing the other variables within the individual functions and reducing their visibility.

If $D = D(A)$ is not connected as an undirected graph, then we regard each connected component of $D$ as a distinct data-diagram $D_i$, each representing a separate algorithm $A_i$ with its inputs and outputs being disjoint from those of the others and with only coincidental cohesion among the $A_i$'s. The equivalence relation "≈" and the various partial orders in Section 2 hold only between data-items in the same $D_i$, and thus the rules DR-1 and DR-2 apply only to each $D_i$ separately. Likewise, the design rule DR-3 given next would also apply to each $D_i$ separately.

## 4. FUNCTIONAL DECOMPOSITION

The next rule DR-3 (for creating functional decomposition) differs from DR-1 and DR-2 significantly in that it groups certain subset of data-items $D' \subset D$ that are connected by one or more links. The simplest such subset is $D' = \overline{in}(d_j)$, which is closely related to the notion of a program-slice [17]. To see this, let $A(d_j)$ be the subset of steps in algorithm $A$ (taken in the order as they appear in $A$) corresponding to the computations $f_{d_i}$ for $d_i \in D' = \overline{in}(d_j)$. Note that $D'$ includes part of $in(D)$ as its own $in(D')$ and, more importantly, $D'$ includes the inputs $in(d_i)$ of each $f_{d_i}$ for $d_i \in D'$. $A(d_j)$ clearly represents a valid algorithm in itself, with its input $D' \cap in(D)$, output $\{d_j\} \cup out(D) \cap D'$, and the data-diagram $D(A(d_j)) = D'$. We can call $A(d_j)$ the *algorithm-slice* for $d_j$. Note that since algorithm $A$ is assumed to be in a high-level form, with each $d_j$ computed in one place in $A$, we do not need to explicitly specify the "position" at which we take the slice for $d_j$, unlike the case for a program-slice. In particular, $A(d_j)$ can be regarded as an encapsulation of some of the initial steps in $A$ and hence it will act only as an initial step if we simplify $A$ by using $A(d_j)$ as a subfunction. For our purpose, we need to consider subalgorithms (subfunctions) of $A$ that are more general in that they might be invoked in an arbitrary place in the simplified form of $A$. We therefore consider more general $D'$ than $\overline{in}(d_j)$. Each $D' = \overline{out}(d_j)$ has properties similar to those of $\overline{in}(d_j)$, except that the subalgorithm associated with $\overline{out}(d_j)$ has the input $in(d_j)$ and it will act only as a terminating step in the simplified form of $A$. Note that we do not have the notion of a program-slice corresponding to $\overline{out}(d_j)$ since we do not typically consider executions of a program starting at arbitrary intermediate points.

### 4.1. Path-Closed Subsets

The following definition characterizes the subset of data-items $D' \subset D = D(A)$ that can be used to form subfunctions of an algorithm $A$. It is a generalization of the notion of a program-slice.

**Def. 4.** A non-empty subset $D' \subseteq D$ is called *path-closed* (or simply, closed) if $D'$ is connected as an undirected graph and for each $d_i, d_j \in D'$ and every path $\pi(d_i, d_j)$ from $d_i$ to $d_j$ in $D$, we have $\pi(d_i, d_j) \subseteq D'$. For our purpose, we will consider for the most part closed subsets that are disjoint from $in(D)$. Since each $\{d_i\}$ and $D$ are trivially closed, the only interesting closed subsets are $D' \subset D$, $|D'| > 1$. □

It is easy to see that $\overline{in}(d_i)$ and $\overline{out}(d_i)$ are path-closed for each $d_i$. The following theorems and corollaries give the structure of path-closed subsets, and show that many familiar subsets $D' \subseteq D$ are path-closed. Theorem 2 does not hold for $D' \cup D''$, in general. Theorem 6 justifies the use of DR-1 and DR-2 before applying the rule DR-3, to be given shortly.

**Theorem 2.** If $D'$ and $D''$ are closed, then $D' \cap D''$ is closed if it is non-empty and connected. □

**Corollary 1.** The subset $D_{ij} = \overline{out}(d_i) \cap \overline{in}(d_j) = \{d_k \colon d_k$ belongs to a path from $d_i$ to $d_j\}$ is path-closed if and only if $d_j \in \overline{out}(d_i)$, or equivalently, $d_i \in \overline{in}(d_j)$. □

**Corollary 2.** The set span$(d_i)$ is path-closed. □

**Theorem 3.** The subset $\overline{in}(d_i) \cup \overline{in}(d_j)$ is closed if and only if it is connected, and $\overline{in}(d_i) \cap \overline{in}(d_j)$ is path-closed if and only if non-empty and connected. Similarly, for $\overline{out}(d_i) \cup \overline{out}(d_j)$ and $\overline{out}(d_i) \cap \overline{out}(d_j)$. □

**Def. 5.** We call $D' \subseteq D$ *in-closed* if it is non-empty, connected (as an undirected graph) and $\overline{in}(d_i) \subseteq D'$ for each $d_i \in D'$, i.e., $D' = \bigcup\{\overline{in}(d_i)\colon d_i \in D'\}$. Similarly, we call $D'$ *out-closed* if it is non-empty, connected, and $\overline{out}(d_i) \subseteq D'$ for each $d_i \in D'$, i.e., $D' = \bigcup\{\overline{out}(d_i)\colon d_i \in D'\}$.

**Theorem 4.** Each closed subset $D'$ can be written as $D' = D'_1 \cap D'_2$, where $D'_1$ is in-closed and $D'_2$ is out-closed. □

Theorem 5 gives an alternative view of path-closed subsets.

**Theorem 5.** Each closed subset $D' \subset D$ can be obtained by successively deleting nodes $d_i$ from $D$ with in-degree or out-degree zero. □

**Corollary 3.** If $D' \subset D$ is closed and $d_i \in in(D')$, then $D' - \{d_i\}$ is closed if it is connected. Likewise, $D' - in(D)$ and $D' - in(D')$ are closed if and only if they are connected. □

**Theorem 6.** If $d_i$ and $d_j$ are as in DR-1 (or DR-2) and $D'$ is a closed subset containing $d_i$, then either $d_j \in D'$ or $D' \cup \{d_j\}$ is closed. □

If we reverse the direction of each link in $D$, then we get another data-diagram, denoted by $rev(D)$. We have $in(rev(D)) = \{d_i\colon in_{rev(D)}(d_i) = out_D(d_i) = \varnothing\} \subseteq out(D)$ and we define $out(rev(D)) = in(D)$. Clearly, a subset of nodes $D'$ is path-closed in $rev(D)$ if and only if it is path-closed in $rev(D)$. It follows from Corollary 2 that *in-span*$(d_j) = \{d_k\colon d_k$ belongs to a path from $d_i$ to $d_j$ for some $d_i$ in $in_D(d_i)\} = span_{rev(D)}(d_i)$ is closed. (The original $span(d_i)$ can be called *out-span*$(d_i)$ in keeping with the notation *in-span*$(d_j)$. This suggest the definition of *in-scope*$(d_j)$ = the complement of $\overline{out}(d_j) \cup \{d_j\}$ = the set of data-items which do not depend on $d_j$ directly or indirectly.)

### 4.2. Function $f_{D'}$

We view a path-closed subset $D'$ in two different ways. First, we regard $D'$ as a whole representing the function $f_{D'}$ (a mega-operation) which computes each $d_i \in D'$ from its respective inputs $in(d_i)$. The inputs of $f_{D'}$ are $in(f_{D'}) = \{d_i \notin D'\colon (d_i, d_j) \in D$ for some $d_j \in D'\}$, which should not be confused with $in(D') \subseteq D'$ which is disjoint from $in(f_{D'})$. Put another way, $D'$ gives only the "body" of $f_{D'}$ and hence $in(f_{D'})$ is not a part of $D'$. Thus several functions may have the same or overlapping inputs, although their "bodies" are necessarily disjoint (because no $d_i$ should be computed in more than one function). In particular, for $D' = \{d_i\}$, this gives $in(f_{D'}) = in(d_i)$, where $f_{D'} = f_{d_i}$. The outputs of $f_{D'}$ are given by $out(f_{D'}) = \{d_j \in D'\colon (d_j, d_k) \in D$ for some $d_k \notin D'\} \cup (D' \cap out(D)) \subset D'$. Unlike $in(f_{D'})$, we have $out(f_{D'}) = out(D')$ when we regard $D'$ as a data-diagram in itself. The function $f_{D'}$ is a generalization of $f_{d_{ij}}$ seen in connection with rules DR-1 and DR-2, except that in the latter there is no link connecting $d_i$ and $d_j$. Note that the subdiagram of $D$ on $D' \cup in(f_{D'})$ should not be taken, in general, as the data-diagram for $f_{D'}$ because the nodes belonging to $in(f_{D'})$ may have links between them.

In the second view, we consider simplifying $D$ by regarding the set of nodes $out(f_{D'})$ as a mega data-item, computed by $f_{D'}$ from the inputs $in(f_{D'})$. We give a more precise description of the simplified form of $D$ next. First, we eliminate the nodes $D' - out(f_{D'})$ from $D$, and add a new node corresponding to the mega data-item for $out(f_{D'})$; we denote the new node also by $D'$ by abuse of notation, but this should not cause any confusion. For each $d_i \in in(f_{D'})$, we add the link $(d_i, D')$; note that $d_i$ is not adjacent to any node in $out(f_{D'})$ in $D$. For each $d_k \in D - D'$ which is adjacent from some $d_j \in out(f_{D'})$, we also add the link $(D', d_k)$. Note that only a part of $out(f_{D'})$ may have been used in $D$ for computing $d_k$, and different $d_k$ may have used different parts of $out(f_{D'})$. To avoid "polluting" the outputs of $D$ due to merging of the nodes $out(f_{D'})$, we require $out(f_{D'}) \subseteq out(D)$ or $out(f_{D'}) \cap out(D) = \emptyset$. The merging of $out(f_{D'})$ into the single node $D'$ is justified in terms of its use in the computations of $d_j \notin D'$ for the case (a) (cf. rule DR-1) or, more generally, for the case (a′) (cf. rule DR-2 or its strengthened form).

(a) For any $d_i$, $d_j \in out(f_{D'})$, $out(d_i) - D' = out(d_j) - D'$.

(a′) For any $d_i$, $d_j \in out(f_{D'})$, $out(d_i) - D' \subseteq out(d_j) - D'$ or vice-versa (or similar conditions with *out* replaced by $\overline{out}$).

**Def. 5.** If $D' \subset D$ is closed, $out(f_{D'}) \subseteq out(D)$ or $out(f_{D'}) \cap out(D) = \emptyset$, and $D'$ satisfies the property (a′) (or, more generally, the property (a′)), then let $D/D'$ denote the reduced data-diagram described above. □

It is easy to see that $D/D'$ is acyclic and $D/\{d_i\} = D$. If we think of each $d_i$-$d_j$ path $\pi(d_i, d_j)$ in $D$ as a potentially distinct semantic dependency of $d_j$ on $d_i$, then for a closed subset $D'$ containing $d_i$ and $d_j$ all such dependencies are accounted for within $f_{D'}$. For each link $(d_i, d_k)$, where $d_i \in D'$ and $d_k \notin D'$, the computation of $d_k$ can proceed only after completion of the computation of $f_{D'}$ and that means there must not be any path from $d_k$ to a node $d_j \in D'$. The closedness of $D'$ assures that this is indeed the case.

**Theorem 7.** If $D'' \subset D'$ are two closed subsets of $D$, then $D''$ as a subset of $D' \cup in(f_{D'})$ is also closed when we consider $D' \cup in(f_{D'})$ as a data-diagram in itself. Moreover, since $in(f_{D''}) \subseteq D' \cup in(f_{D'})$, we can regard $f_{D''}$ as a subfunction of $f_{D'}$. □

### 4.3. Design Rule DR-3

We form a functional decomposition of $D = D(A)$ by choosing a family of disjoint path-closed subsets $D' \subset D$ and applying rule DR-3 to each of them. To form a functional decomposition of $f_{D'}$ itself, we apply the rules {DR-1, DR-2} afresh to $D' \cup in(f_{D'})$ since this may allow additional merging of nodes within the subdiagram $D' \cup in(f_{D'})$.

**Rule DR-3:** If $D' \subseteq D$ is closed and the conditions in Def. 5 hold, then form the function $f_{D'}$ associated with $D'$.

**Example 4.** Fig. 3(ii) shows two closed sets $D' = \{d_1, d_{2345}, d_6\}$ and $D'' = \{d_{78}, d_9, \text{output}\}$ in Fig. 3(i) which satisfy the conditions for the application of rule DR-3. In particular, $out(d_6) - D' \subseteq out(d_{2345}) - D'$ and we have $out(f_{D'}) \cap out(D) = \emptyset$. The link (Input, $D'$) in Fig. 3(ii) represents a call to the function Graph() $= f_{D'}$, which carries out the computations of $d_1$, $d_{2345}$, and $d_6$ in Fig. 3(i). The link $(D', D'')$ represents a call to the function NewImage() $= f_{D''}$, which carries out the computations of $d_{78}$, $d_9$, and *Output* in Fig. 3(i). There are many other closed sets in Fig. 3(i). The closed sets $D'$ and $D''$ give the following functional decomposition of H-CONVEX, where we show the inputs and outputs of each function as two sets of data-items for notational convenience, with the input-set shown on the left of the function and the output-set on the right.

$$\begin{array}{l} \langle\{\text{input}\}\ \text{H-CONVEX()}\ \{\text{output}\}\rangle = \\ \quad \langle\{\text{input}\}\quad \text{Graph()}\quad \{d_{2345}, d_6\}\rangle; \\ \quad \langle\{d_{2345}, d_6\}\ \text{NewImage()}\ \{\text{output}\}\rangle. \end{array}$$

The function NewImage() $= f_{D''}$ above has two local variables $d_{78}$ and $d_9$, and two input variables $\{d_{2345}, d_6\}$, which are the output of the function Graph() $= f_{D'}$. The merging of $d_{2345}$ and $d_6$ into a single data-item (cf. Def. 5) reduces both the outputs of Graph() and the inputs of NewImage() to a single data-item $d_{23456}$.

This reduces the number of global variables, other than the input and output (and $M$, $N$, and numSegments), from 5 in Fig. 3(i) to 1 for Fig. 3(ii). The visibility of $d_1$ is limited now to within Graph(), by making it a local variable for Graph(), and similarly the visibility of $\{d_{78}, d_9\}$ is limited to within NewImage(). □

There are several other functional decompositions which also require only one global variable. For example, we could remove $d_{78}$ from $D''$ and put it in $D'$. The new $f_{D'}$ will now have $d_6$ as a local variable $d_{78}$ would become part of its output. This is somewhat less desirable than the one in Example 4 because neither of $out(d_{78}) - D'$ and $out(d_{2345}) - D'$ is now a subset of the other; instead, we only have $\overline{out}(d_{2345}) - D' \subseteq \overline{out}(d_{78}) - D'$. An implementation that supports both of these functional decompositions is a structure that combines all of $\{d_{2345}, d_6, d_{78}\}$; some parts of which will be used for output of one function and some parts of which it will be used locally by the other function. There is one drawback in this design, however, in that the components $d_6$ and $d_{78}$ become visible more than necessary.

We avoided merging $d_{2345}$ and $d_6$ into $d_{23456}$ till now to prevent loops in the data-diagram. However, in terms of the use of $d_{2345}$ and $d_6$ outside of Graph() they can be merged, and for Graph() itself the computation of $d_6$ related to the links $\{(d_{2345}, d_6), (d_1, d_6)\}$ is viewed now as modifying the component $d_6$ within $d_{23456}$. We do not merge $d_1$ with $d_{2345}$ (by DR-1 or DR-2) even within the data-diagram for Graph() because one of them is an output and the other is not; the creation of $d_{12345}$ would also cause an unnecessary visibility of the part $d_1$ outside Graph(). If $d_4$ and $d_5$ were not merged with $d_{23}$ in Fig. 3(i), they would now become output nodes of Graph() and equivalent to $d_{23}$ for their use outside Graph(), justifying their merging with $d_{23}$. The above discussion shows the need to delay the application of DR-2 as much as possible until the functional decomposition is obtained.

One reason for not putting both $d_{78}$ and $d_9$ in $D'$ above (and therefore making $d_{78}$ a local variable in Graph()) is that one can expand algorithm H-CONVEX for other purposes where $d_{78}$ is needed together with $d_6$ in those computations.

### 4.4. Case of $|in(d_j)| = 1 = |out(d_j)|$

If $|in(d_j)| = 1 = |out(d_j)|$, it is easy to create closed sets containing $d_i$. Let $(d_i, d_j)$ be the only link to $d_j$ and $(d_j, d_k)$ the only link from it. First, both $D'_1 = \{d_i, d_j\} = \overline{in}(d_j) \cap \overline{out}(d_i)$ and $D'_2 = \{d_j, d_k\} = \overline{in}(d_k) \cap \overline{out}(d_j)$ are closed sets. Also, if $D'$ is a closed set such that $d_i \in D'$ and $d_j \notin D'$ (and hence $\overline{out}(d_j) \cap D' = \varnothing$), then $D' \cup \{d_j\}$ is a closed set; similarly, if $d_j \in D'$ and $d_i \notin D'$ (and hence $\overline{in}(d_i) \cap D' = \varnothing$), then $D' \cup \{d_i\}$ is a closed set.

### 4.5. Role of Span($d_i$)

If we use a closed set $D'$ in a functional decomposition and $d_i \in D'$ but span($d_i$) $\not\subseteq D'$, then $d_i$ must be an output for $f_{D'}$ and hence it needs to be a global variable. Put another way, span($d_i$) is the smallest body of a function containing $d_i$ if $d_i$ is not a global variable. To reduce the number of global variables, it is therefore desirable that each $D'$ used in a functional decomposition contains span ($d_i$) for as many $d_i \in D'$ as possible. If span($d_i$) $\subseteq D'$, then $f_{D'}$ accounts for all necessary uses of $d_i$. For some $d_i \in D'$, we may be forced to include only a part of span($d_i$) in $D'$ to avoid making $D'$ and hence $f_{D'}$ too large. In Fig. 3(i), span($d_{2345}$) is the only span that has size > 2 and it is in fact too large, which justifies making it a global variable by not including all of span($d_{2345}$) within one of the functions in the decomposition shown in Fig. 3(ii). If we formed a function using span($d_{2345}$) = $\{d_{2345}, d_5, d_{78}, d_8, \text{output}\}$, then $d_1$ will be a global variable but span($d_1$) has size only 2. In contrast, the functional decomposition shown in Fig. 3(ii) makes $d_1$ a local variable in $f_{D'}$ and eliminates the visibility of $d_1$ outside $f_{D'}$.

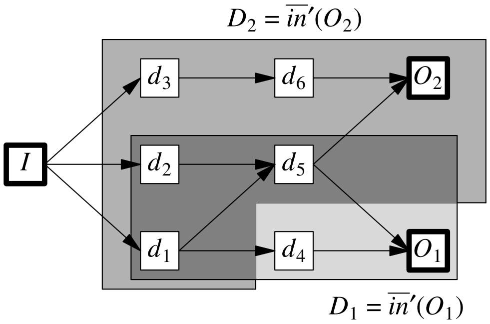
## 5. TOP-DOWN DECOMPOSITION USING PATH-CLOSED SUBSETS

As we remarked earlier, path-closed subsets $D' \subset D$ can be used effectively for a top-down functional decomposition. Once the larger functions $f_{D'}$ are identified, a subfunction of $f_{D'}$ is given by a closed subset $D'' \subset D'$. This approach eliminates the traditional problem of "making commitments to systems structure from a position of ignorance" [6] because of our use of the complete data-flow diagram at hand. Within the top-down approach itself, we can use more specifically a goal or output-oriented design as illustrated below. The following theorem forms the basis of the output-oriented design approach. Recall that closed subsets represent the body of functions, exclusive of their inputs.

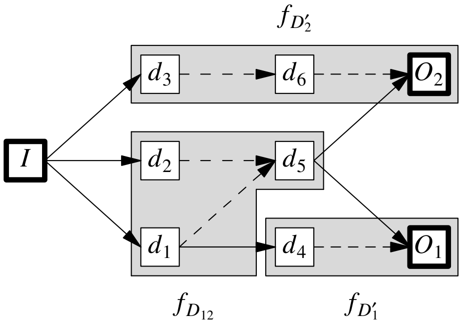
**Theorem 8.** For any $d_i \notin \overline{in}(d_j)$, we have $\overline{in}(d_i) - \overline{in}(d_j)$ is a closed subset. The same is true for $\overline{in}'(d_i) - \overline{in}'(d_j)$, where $\overline{in}'(d_i) = \overline{in}(d_i) - in(D)$. □

**Example 5.** Consider the data-diagram shown in Fig. 4(i). Here, $D_1 = \overline{in}'(O_1) = \{d_1, d_2, d_4, d_5, O_1\}$ and $D_2 = \overline{in}'(O_2) = \{d_1, d_2, d_3, d_5, d_6, O_2\}$ are closed as are $D'_1 = D_1 - D_2 = \{d_4, O_1\}$, $D'_2 = D_2 - D_1 = \{d_3, d_6, O_2\}$, and $D_{12} = D_1 \cap D_2 = \{d_1, d_2, d_5\}$ (cf. Coro. 3, Theorem 8, and Theorem 2). Fig. 4(ii) shows the functional decomposition based on $D'_1$, $D'_2$, and $D_{12}$. Note that $f_{D_{12}}$ is a subfunction of $f_{D_1}$, and $f_{D'_1}$ is the remainder of $f_{D_1}$ when we subtract $f_{D_{12}}$ from it; a similar remark holds for $f_{D_2}$ and $f_{D'_2}$. We cannot merge $d_1$ and $d_2$ by

DR-2 applied o $f_{D_{12}}$ because only one of them is an output of $f_{D_{12}}$ (although they could be merged with respect to $D$ by DR-2 even if we use its strengthened form). This shows once again that application of DR-2 should be delayed as far as possible. We can merge $d_1$ and $d_5$ by DR-2 applied to $f_{D_{12}}$ (and also to $D$), giving only one global variable $d_{15}$ for the functional decomposition. If $D_{12}$ were not connected, then each connected component of it would give a function, which would be applied before applying $f_{D'_1}$ and $f_{D'_2}$. The functional decomposition consisting of $f_{D_1}$ and $f_{D'_2}$ also require only one global variables ($d_5$). □




(i) $D_{12} = D_1 \cap D_2 = \overline{in}'(d_5)$ is closed.




(ii) The functional decomposition based on $D'_1 = D_1 - D_2$, $D'_2 = D_2 - D_1$, and $D_{12}$ in (i)

Figure 4. Illustration of output (goal) oriented, top-down functional decomposition.

Consider now rev($D$) of the data-diagram in Fig. 4(i). To apply the above method, we momentarily use $\{d_1, d_2, d_3\}$ as the output variables instead of $\{I\}$. This gives rise to the closed sets $\{d_1, d_4\}$, $\{d_2\}$, $\{d_5\}$, and $\{d_3, d_6\}$. We can form a larger closed set by including $d_5$ in $\{d_2\}$ or in $\{d_1, d_4\}$. This still does not give as good a functional decomposition as the one in Fig. 4(ii). This should not surprise us because the role of inputs and outputs in a data-diagram is not symmetrical, and therefore output-oriented design and input-oriented design (= output-oriented design using rev($D$)) are not expected to have similar properties. We can use any subset of nodes which form a cut set to separate the inputs from the outputs in the role of outputs in the output-oriented design.

TABLE 2. An algorithm corresponding to Fig. 4(i) and its restructured form based on Fig. 4(ii).

| **Algorithm** $A$ |
|---|
| 1. Compute $d_1$, $d_2$, and $d_3$ from $I$. |
| 2. Compute $d_4$ from $d_1$, $d_5$ from $\{d_1, d_2\}$, and $d_6$ from $d_3$. |
| 3. Compute $O_1$ from $\{d_4, d_5\}$ and compute $O_2$ from $\{d_5, d_6\}$. |
| **Restructured form of** $A$ |
| 1. Compute $d_{15}$ from $I$ by applying $f_{D_{12}}$ to the input $\{I\}$. |
| 2. Compute $O_1$ by applying $f_{D'_1}$ to the inputs $\{d_{15}\}$. |
| 3. Compute $O_2$ by applying $f_{D'_2}$ to the inputs $\{I, d_{15}\}$ (using only the part $d_5$ in $d_{15}$). |

## 6. MANY PATH-CLOSED SUBSETS

Consider the data-diagram $D$ in Fig. 5, which has $N+2$ nodes and a linear $O(N)$ number of links. Here, each of $n_1$, $n_2 - n_1$, $\cdots$, $n_k - n_{k-1}$ and $k$ is approximately √N and $n_k = N$. Choose $i_j$, $1 \le j \le k$, such that $n_{j-1} < i_j \le n_j$ ($n_0 = 0$). The union of any subfamily of $\{\overline{out}(d_{i_1}), \overline{out}(d_{i_2}), \cdots, \overline{out}(d_{i_k})\}$ then gives a closed subset. Similarly, if we replace $\overline{out}(d_{i_j})$'s by $\overline{in}(d_{i_j})$'s. We can also form closed subsets by taking the union of some $\overline{in}(d_{i_j})$ with some other $\overline{out}(d_{i_j})$ as long as we keep the union connected. This gives an exponentially large number of closed subsets $D'$ with $out(f_{D'}) \subseteq out(D)$ (cf. Def 5.).

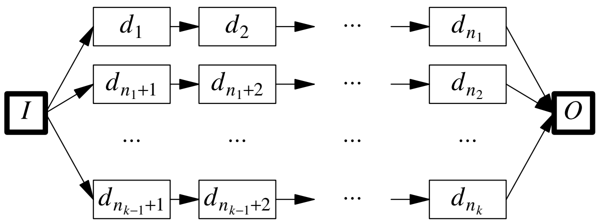



Figure 5. A data-diagram with an exponentially large number of path-closed subsets.

The existence of many path-closed subsets in a data-diagram $D$ means one has many possible choices in creating a functional decomposition. We therefore need an efficient method for determining an optimal decomposition of $D$ into $k \ge 1$ disjoint path-closed subsets $D'_i$, which minimizes the visibility of variables, say. One can add other constraints in terms of the size $|D'_i|$ (or, more generally, estimated total size of $f_{d_j}$'s for $d_j \in D'_i$). If the implementation language allows nested function definitions, one may allow $D'_i$'s to be nested, and also put an upper limit on their nesting depth. In any case, creating many functions, each corresponding to a small closed subset $D'_i \subset D$ is typically not helpful because that simply replaces the problem of many global variables by that of many functions. We leave these optimization issues as a future research task. The example below illustrates the minimization of visibility of variables for a simple data-diagram $D$ and $k = 2$.

**Example 6.** Consider the data-diagram $D$ in Fig. 6 and its functional decomposition indicated by the two ($k = 2$) dashed boxes. The only global variable is $d_m$, other than the input $I$ and the output $O$. The function for $D_1$ eliminates the visibility of its local variables $\{d_i\colon 1 \le i \le (m-1)\}$ to the local variables in the function for $D_2$. The visibility of the output $d_m$ of the functions for $D_1$ is not reduced. The total reduction in visibility is $(m-1)(n-m)$, which is minimum when $m = \lceil n/2 \rceil$. Similarly, for $k = 3$, the optimal decomposition breaks the chain in three approximately equal parts. □

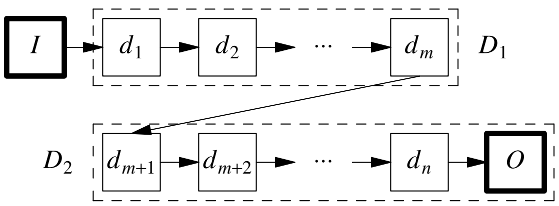



Figure 6. A functional decomposition of a simple data-diagram into two parts.

## 7. OTHER ISSUES

The design rules {DR-1, DR-2, DR-3} given here focus only on the abstract structure of the data-diagram $D$ of an algorithm. They do not address, in particular, other issues in software design that are not captured directly by the data-diagram.

A software design needs to address the issues related to testing and future extensions as well. It may be desirable to choose a design that facilitates these aspects at the cost of less than minimal visibility of various variables. In the case of algorithm H-CONVEX, a possible extension is the alternate use of segment sizes (size[$k$]) in defining a different notion of the path length and maybe even a different notion of best segment. In that case, it would be appropriate to break up the closed set $D'' = \{d_{78}, d_9, \text{output}\}$ in Fig. 3(ii) into two parts $D''_1 = \{d_{78}, d_9\}$ and $D''_2 = \{\text{output}\}$, which would increase the global variables by one (from $\{d_{23456}\}$ to $\{d_{23456}, d_9\}$). This would allow testing the functions $f_{D''_1}$ and $f_{D''_2}$ associated with $D''_1$ and $D''_2$ more easily than testing of $f_{D''}$.

### 7.1. Relationship To Object-Oriented Approach

For computing a largest $h$-convex subset of an image (as in algorithm H-CONVEX), the usual notion of physical objects present in the image plays no role and we cannot use those objects in designing the algorithm or the software. (Also, the methods for identifying such objects can be very complex and computationally expensive, compared to the complexity $O(MN)$ of the algorithm H-CONVEX in Appendix for an $M \times N$ image.) The rows or columns or single pixels in the image cannot serve as a suitable notion of an object either. In fact, a little thought shows that the only suitable notion of an object here is an $h$-segment, because of the property that each h-segment $s$ is either contained in the new image or $s$ is totally disjoint from it. Other similar notions, like vertical segments in the image or vertical or horizontal segments in the complement of the image, do not serve any purpose here.

The appropriate notion of an object in the object-oriented approach for software design can be thus determined only from the details of the algorithm and not from any preconceived notions based on the problem statement (input and output). In particular, we feel that "objects" in an implementation can be designed only after obtaining the functional decomposition of an algorithm. This concurs with the experimental studies [15] which show that functional decomposition is more effective than object-orientation in client/developer communication in early stages of the software development and with other studies [1, 12, 20] which show that functional decomposition (and data-flow diagrams) are easier to learn. If the program-objects turns out to be related to the physical-objects involved in the problem at hand, then it only means that the algorithm itself is structured around those physical objects, which is often the case in business data processing, for example. But the absence of such relationship between physical-objects and program-objects by no means indicates in itself an improper design of the software.

## 8. CONCLUSION

We have shown that the path-closed subsets of a data-flow diagram play an important role in software functional design by identifying the groups of data-items and the operations associated with them to form the functions. We have shown that the design process can be simplified by using a certain equivalence relation (and to some extent using a related partial ordering) among the data-items. It is our hope that this approach to functional design can automate a good part of the functional design task given the data-flow diagram of an algorithm. We have demonstrated the usefulness of our method for an algorithm involving complex data-flows and a variety of data-structure. We believe that our ideas can be used for both small and large softwares. One of the future tasks is to determine efficient algorithms for an optimal functional design that minimizes the unnecessary visibility of data-items, the number of global variables (data-items), etc.

## 9. REFERENCES


1. Agarwal, R., De, P., and Sinha, A.P., Comprehending object and process model: an empirical study, *IEEE Transactions on Software Engineering,* 25(1999), pp. 541-556.
2. Beizer, B. *Software testing techniques,* Van Nostrand Reinhold Comp, New York, 1983.
3. Bieman, J.M. and Ott, L.M., Measuring Functional Cohesion, *IEEE Transactions on Software Engineering,* 29(1994)8, pp. 644-657.
4. Bogdan Korel and Janusz Laski, Dynamic program slicing, *Information Processing Letters,* 29(3):155-163, October 1988.
5. Blum, B.I., *Software engineering: a holistic view,* Oxford University Press, New York, 1992.
6. Blum, B.I., A taxonomy of software development methods, *Communications of ACM,* 37(199), pp. 82-94.
7. Card, D.N., Church, V.E., and Agresti, W.W., An Empirical Study of Software Design Practices, *IEEE Transactions on Software Engineering,* 12(1986), pp. 264-271.
8. Card, D.N., Page G.T., and McGarry, F.E., Criteria for Software Modularization, *Proceedings of the 8th International Conference on Software Engineering,* August 28-30, London, 1985, pp. 372-377.
9. DeMarco, T., *Structured analysis and system design, Yourdon Inc., New York, 1979.*

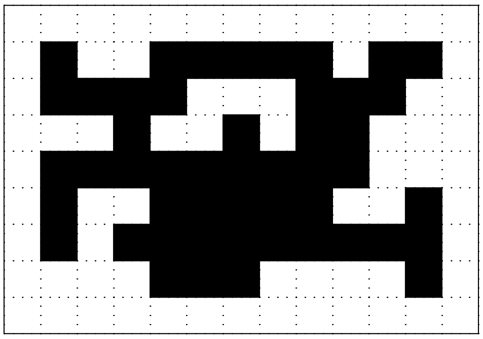
10 Dhama, H., Quantitative Models of Cohesion and Coupling in Software, *Proceedings of the Annual Oregon Workshop on Software Metrics,* April 10-12, 1994, Silver Falls, Oregon.

11. Frank, T., A survey of program slicing techniques, *Journal of Programming Languages,* 3(1995), pp. 121-189.
12. Gemino, W. and Wand, Y., Empirical comparison of object-oriented and data-flow models, *Proc. 18th International Conf. Information Systems,* pp. 446-457 1997.
13. Ghezzi, C., Jazayeri, M., and Mandrioli, D., *Fundamentals of software engineering,* Prentice-Hall, Inc., USA, 1991.
14. 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.
15. Moynihan, T., An experimental comparison of object oriented and functional decomposition paradigms for communicating to users, *J. Systems and Software,* 33(1996), pp. 163-169.
16. 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.
17. 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.
18. Stevens, W.P., Myers, G.J., and Constantine, L.L., Structured design, *IBM Systems Journal,* 13(1974), pp. 115-139.
19. Storer, R., *Practical program development using JSP,* Blackwell Scientific Publ, USA, 1987.
20. Vessey, I. and Conger, S.A., Requirement specification: learning object, process, and data methodologies, *Communications of ACM,* 37(1994), pp. 102-113.
21. Weiser, M., Program slicing, *IEEE Trans. Software Engineering,* SE-10, 4(1984), pp. 352-357.

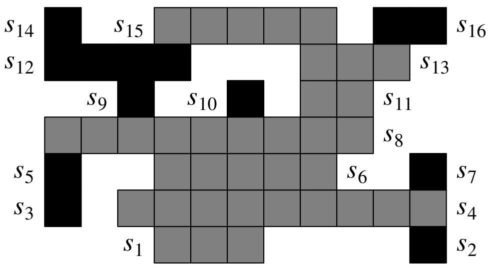

# APPENDIX: an algorithm with complex data-structures and data-flows

We briefly describe here an image processing algorithm, involving complex data-structures and data-flows, which is used in the paper to illustrate the new software-design method.

We consider here black-and-white images, which are made of unit squares (pixels); the black pixels constitute the image and the white pixels constitute the background. See Fig. A-1(i). An important technique in image analysis is to consider subimages of an image having a given structure or shape. One such structure corresponds to the notion of a *horizontally convex* (in short, *h*-convex) subset. A subset of an image is called *h*-convex if for each pair of pixels in the subset that are in the same row (horizontal line), all pixel between them are also in the subset. The image in Fig. A-1(i) is not *h*-convex because the second row has three missing pixels between the leftmost and the rightmost pixels of the image. Two pixels are *adjacent* if they have a common horizontal or vertical side; an image is *connected* if each pixel in it can be reached from any other pixel in the image by a path connecting adjacent pixels. An *h-segment* of an image is a maximal set of consecutive pixels in the same row. The size of an image or an $h$-segment is the number of pixels in it. The image in Fig. A-1(i) has sixteen $h$-segments and they are shows as $s_1$ to $s_{16}$ in Fig. A-1(ii). An image is $h$-convex if and only if there is at most one $h$-segment in each row. For each $h$-segment $s_i$, a maximal $h$-convex subimage of an image either contains all of $s_i$ or is disjoint from $s_i$. The shaded pixels in Fig. A-1(ii) show a subimage of the image in Fig. A-1(i) which is $h$-convex, connected, and has the largest size. Another such subimage is obtained by replacing the segments $\{s_{11}, s_{13}\}$ by $\{s_9, s_{12}\}$.



(i) A black and white image.




(ii) The sixteen $h$-segments are $s_1$ to $s_{16}$ and the shaded part is a largest connected, $h$-convex subimage.

Figure A-1. Illustration of a largest connected, $h$-convex subimage.

**Algorithm H-CONVEX():**

**Input:** An $M \times N$ matrix image$[i, j]$ for a binary image, which may or may not be connected.

**Output:** The $M \times N$ matrix newImage$[i, j]$ corresponding to a largest connected, $h$-convex subimage of image$[i, j]$.

1. [Find $h$-segments.] For (each row $i = 1, 2, \cdots, M$), find the list segList($i$) of $h$-segments in that row. Also, for each segment $s_k$, compute left($s_k$) = the column number of the leftmost pixel in $s_k$, right($s_k$) = the column number of the rightmost pixel in $s_k$, and size($s_k$) = size of $s_k$. Let row($s_k$) = $i$ for each $s_k \in$ segList($i$).

2. [Find adjacencies among $h$-segments.] For (each $i = 1, 2, \cdots, M-1$) do the following:

   (a) Let adj($s_k$) = empty-list for each $s_k \in$ segList($i$). Also, let $s$ = the first item in segList($i$) and $s'$ = the first item in segList($i+1$).

   (b) Repeat the following until $s$ or $s'$ is non-existent, i.e., the corresponding list is exhausted.

   If (right($s'$) < left($s$)) then let $s'$ be the next item in segList($i+1$)

   else if (right($s$) < left($s'$)) then let $s$ be the next item in segList($i$)

   else add $s'$ to adj($s$), let $s'$ be the next item in segList($i+1$) if right($s'$) $\leq$ right($s$), and let $s$ be the next item in segList($i$) if right($s'$) $\geq$ right($s$).

   (c) Let adj($s_k$) = empty-list for each $s_k \in$ segList($M$).

3. [Find longestPathLengths.] Apply the longest-path algorithm to the acyclic digraph given by adj($s_k$)'s, with size($s_k$) being the link-weight for each arc to the $h$-segment $s_k$, and compute longestPathLength($s_k$) and parentInPath($s_k$) for each $h$-segment $s_k$. (A longest path to $s_k$ is obtained by tracing the parents backwards: $s_k$, parentInPath($s_k$), parentInPath(parentInPath($s_k$)), etc.)

4. [Find a longest path.] Let bestSegment = $s$, where longestPathLength($s$) has the largest value, and let $\pi$ = a longest path to bestSegment.

5. [Create the output image.] Form the newImage consisting of the $h$-segments $s_j \in \pi$, using left($s_j$) and right($s_j$) for the range of columns for pixels in $s_j$ and using row($s_j$) for the row number of these pixels.