Data accessibility in structured programming

by NED CHAPIN
Data Processing Consultant
InfoSci Inc.
Menlo Park, California

INTRODUCTION

C. A. R. Hoare, N. Wirth, E. W. Dijkstra, and some others active in structured programing have described and tacitly recommended a hierarchical accessibility of passed data in structured programing (see for example, References 11 and 26). This fits well, they and others indicate, with both the theory and the practice of structured programing (see for example, References 15, 18, 19, and 25).

This paper does not question that the hierarchical accessibility with parameter passing can be made to work. It may even be the most elegant data accessibility for some interesting academic problems. However, this paper does argue that for most practical software design situations, other approaches are on balance superior, especially for the kinds of situations commonly implemented using COBOL, FORTRAN, PL/I, JOVIAL, or BASIC. To support this argument, this paper presents first some definitions of terms, and then an examination of some evidence in nine categories. Following a ranking evaluation, the paper draws a conclusion.

DEFINITIONS

For purposes here, a system or program designed and implemented with structured programing techniques is characterized as having a proper nesting of functions in three basic control patterns with single entrance and exit, in the sense defined by Mills.20 This paper does not use “structured programing” to mean “structured coding,” but rather uses it in the sense sketched by Dijkstra.5 Further, structured programing is characterized by a hierarchical structure of modules (segments) again in the sense defined by Mills, which to people appear to specify the performance of specific single functions, as defined by Parnas and McGowan.19,22 The function parsing techniques that lead to the creation of such proper-nested structures have been described elsewhere.8 The object is ease of human understanding.7,16

For the purposes of this paper, a level of abstraction as it applies to data is defined to be a grouping of data based upon the recognition by people of some common characteristics, often semantic rather than structural.11,19,24 Data conforming to levels of abstraction is referred to in this paper as “leveled data.” For example, a file (composed of records in turn composed of fields) is regarded as more abstract than the component fields individually. In contrast, a disjoint string is defined to be a string data structure which has as member elements items of data with no necessity of similarity in level of abstraction. Thus a disjoint string might be composed of two Boolean variables, a record from a file, three tables, and a queue. Pointers may be among the elements of a disjoint string.17 Because of the potential diversity, each member element in the string is specifically enumerated and described.

For the purposes of this paper, a flow graph is defined to be a graphic presentation of part or all of a control flow in a system or program.1 The edges in the graph represent an explicit flow of control unaccompanied by the manipulation of data but accompanied by the implicit or explicit communication of data. The nodes in the graph represent an implicit flow of control accompanied by successive steps in the explicit manipulation of data, and accompanied by some means for the communication of data among the steps.

EXAMINATION

Access volume potential

The access volume potential is a count of the maximum possible number of data items with assigned values which are accessible to any given module upon invocation, summed for all modules. Ideally, this count should be small, since the larger it is, the greater is the difficulty of human understanding in design, implementation, and maintenance.

The calculation of the access to data items has been described by Allen and by Cocke.1,2 Following “procedure A” as an “algorithm for finding intervals,” a partition can be made of a control flow graph into basic intervals. If the hierarchy of modules is a true tree conforming to a proper nesting of functions, then it is possible to reduce selectively the basic control graph to yield nested intervals of higher and higher order, as diagramed in Figure 1. These nested intervals can correspond not only to the modules, but also to the levels in the hierarchy, since proper nesting in the hierarchy yields a reducible control flow graph.19 Such a selectively applied flow-graph reduction is roughly equiva-
Because of this correspondence, the control flow graph can be used as a basis for determining the intrasystem access to data, given a structured programming design, and implementation. With global data, the “live” access potential \( (L) \) for any one module is given by the union of (a) the union \( (U) \) of the assignments \( (AN) \) and definitions \( (DN) \) of data within the module, with (b) the union of the data assignments and definitions available on the entrance edge of module \( i \) that do “reach” \( (R) \) that module:

\[
L_i = R_i U(AN_i UDN_i) \quad (1)
\]

Note that the questions of whether or not the module \( i \) preserves or changes the definition of assignment, or whether or not data so specified are used in the module is irrelevant in determining the live access potential at any one module. If modules be identified in execution sequence, then the reach \( (R) \) expands as the control flow covers more modules. Thus, the access potential increases with each module executed by the new assignments \( AN \) and new definitions \( DN \) effective in each module.

If \( \rho \) indicates a count function, then the volume or amount of live access potential \( (A) \) in a program or system of \( n \) modules with global data communication is estimated by:

\[
A = \sum_{i=1}^{n} \rho L_i \quad (2)
\]

With passed data, the amount of live access potential is much smaller than for the global case with the use of structured programming techniques. The live access potential is given by expression (1) as before, but now the set \( R_i \) is much reduced because it is a subset of the global data case, limited to the data deliberately communicated between the \( i \)th modules and other modules. This also keeps expression (2) much smaller.
With leveled data, the minimum total number of data items assigned values (AN) or defined (DN) is never smaller, and is usually larger, than the corresponding values for the disjoint strings. This is because the disjoint strings make provision only for data items actually accessed, whereas leveled data may include data defined or assigned for consistency, elegance, or symmetry of conception.

Expressed as a ranking, with 1 for the most desirable, and 4 for the least, passed disjoint strings are of rank 1, and global leveled data are of rank 4. This is summarized as the top row in Table I.

Access values

The access to data values treats each item of data as a different item of data when it takes on a different value. A count of these is an indicator of the diffuseness of function in the hierarchy and of the difficulty of debugging and maintaining a system of program.

The estimating procedures for access to values used for global data has been well presented by Allen and Cocke. Domains are difficult to specify from the flow graph and from the hierarchy. In practice, the availability of input-output tables is helpful.

With passed data, the access to data values is much restricted, and domains for data entering a node from along any edge of the control flow graph can be cleanly specified. In terms of data values used in the module, the access is the same as for the global case noted above. But to this must be added the access to values arising from the communication of the data through the module. Much data may have to be passed through some modules unchanged in value or identity in a hierarchical structure. Diagramed in Figure 2 is a situation where an item of data is assigned an identity and value in module X and accessed in module Y to be modified in value but not identity for use in module Z. The communication from X to Y involves a pass-through chain containing three pass-through nodes, and from Y to Z of two pass-through nodes.

With leveled data, the count of the data values accessed is greater than for the disjoint string, because extraneous values are presented in a module at levels above the lowest in the hierarchy. Hence, the ranking for access to values as summarized in Table I is best for the global disjoint string, and worst for the passed leveled data.
Congruence

When the pattern of the data communication among the modules of the hierarchy has a high degree of congruence with the pattern of the flow of control, design, implementation, and maintenance work is easier. People only have to know one pattern to understand two aspects of the program or system. The use of input-output tables helps reveal the degree of congruence between data and control flow.\(^7\)

Global data offers little congruence with the nested hierarchy of the tree, as diagramed in Figure 3. This is the case for both leveled data and disjoint strings. This is because the data flow is not constrained to match the control flow, and in practice is usually a network, not a tree if the modules be treated as nodes for data flow.\(^8\)

Passed data offers excellent congruence with the nested hierarchy of the tree, as diagramed in Figure 4. This is the case for both leveled data and disjoint strings, but the disjoint string reduces the presence of unused data. The ranking, therefore for congruence places the passed disjoint string in the best position, and gives a tie for the two alternatives in the global data case, as shown in Table 1.

Coupling

While the concept of coupling comes from structured design, it can be applied to software designs done with structured programing. Using the definition and scale of coupling as laid out by Myers, the global data case is at best common coupled.\(^\text{21}\) This scale position is the same for both leveled data and for disjoint strings.

The passed data case requires a closer inspection. Using leveled data results in stamp coupling except sometimes at the leaf position in the hierarchy. Using disjoint strings enables the consistent use of elementary items of data throughout the hierarchy, and results in the more desirable data coupling between modules on adjacent levels vertically in the hierarchy. In either situation, no coupling at all characterizes all of the other coupling relationships between modules. In short, the ranking of coupling is the same as for congruence, as noted in Table 1.

Strength

While the concept of strength comes from structured design, it can be applied to software designs done with structured programing. Using the definition and scale of strength as laid out by Myers, the evidence on strength is moot.\(^\text{21}\) Sometimes one alternative seems to go with higher strength and sometimes not. Hence, the ranking for strength is a tie for all, as shown in Table 1.

While not evidence, an observation may add insight. Since in structured programing, the focus is on the function of a module, the data flow into and from a module serves effectively as a definition of that function. For high strength, a module must require all of its input to yield its outputs, and not be partitionable into two or more parts with no increase in the number of intermodule items of data needed, ignoring pass-throughs. Therefore, a partitioning of functions implies a partitioning also of data flow if high strength is to be maintained, as diagramed in Figure 5.\(^\text{6,8}\) The means of achieving that data flow is really an independent question. In a high-strength design, the actual access of data for use is generally closely similar whether the design or the implementation be passed data or global, but the "pass-through unchanged" pattern is very different.

Overhead

Execution overhead in an implemented program or system depends on many factors, including the computer configuration, the language, the compiler, the operating system, the shop practices, etc. Even if attention be given only to two aspects, such as CPU cycles used, and amount of internal storage committed, only general trends can be described. No firm consistent evidence is available, or probably even possible beyond an instance by instance basis.

Global data are the most economical of storage space and fastest in execution speed if disjoint strings be handled. Both measures are impaired by using levels of abstraction because of the increased size of the units of data handled. This is reflected in the rankings shown in Table 1.

![Figure 5—Diagram of some data implications of high strength](From the collection of the Computer History Museum (www.computerhistory.org))
Passed data are still less economical and slower, again in the same two steps, because of the increased handling of data that are involved. The higher the strength of the modules, the smaller is the impairment because the more elegant is the use of data. But the smaller the modules the larger is the impairment, because calling and passing becomes a larger proportion of the total work to be done. This degrades the ranking for the passed data alternatives, as shown in Table I.

**Human convenience**

Both the global and the passed data are convenient for people, but in different ways. The global data offers single description (declaration), economy of effort, and ease of reference to the description for people. But global data also offer the least visibility into the steps in the use, transformation, and production of the values of the data—a critical matter in a function-oriented approach.

The passed data offer uniformity in conventions and high visibility. But the cost is in redundancy and repetition, both in the designs and in the code. At the compile stage the language conventions provide an automatic check on human performance. This enforcement helps keep the data visibility high.

Convenience is also both aided and hurt both by leveled data and by disjoint strings. The leveled data provide an easily comprehended grouping of data but hide data identities—a serious matter in debugging and maintenance. The disjoint strings lack the mnemonic quality that aids the human memory, but do focus attention on actual functions.

A ranking of the human convenience of necessity is subjective, and reflects value judgments about operating environments as well. An informal survey conducted over half a year’s time involving persons who design computer software and try to use structured programing techniques, and including people from all across the continent, indicated the preferences shown in Table I on convenience.

**Maintainability**

Provided the visibility is good, the reports from people doing the work seems consistent on maintainability: the passed data in disjoint strings has the lowest cost of maintenance. Some of this comes from the access measures noted earlier, some from congruence, and some from coupling. People can follow the items of data more easily, and the functions are usually more clearly specified and understandable with the passed data in disjoint strings. By contrast, the leveled data have to be peeled away to specifics, and hence involve more maintenance effort.

For these same reasons, the global data offer less aid in maintenance. The value changes in the data are more difficult to follow, especially when leveled data are used. In the absence of countermeasures, it takes people longer to find out what is going on. Hence, in the summary ranking, the passing of disjoint strings is shown as a 1, and the leveled global data is shown as a 4.

**System quality**

System quality has many measures. One is number of bugs encountered over some period of time. Using techniques described by Haney and Myers, it is possible to give a quantitative tinge to measuring system quality in terms of bugs.

Applying those measures to examples drawn from industrial and governmental computer applications, indicates that passed data in disjoint strings has the lowest expectation of bugs. Passed leveled data has an expectation about one-third higher. Global data in either form yield an expectation for nearly double the number of bugs. This gives a tie in the ranking shown in Table I.

Criticism of these techniques and the use of bug counts as quality measures has been heard. Both are admittedly imprecise—but they are also the best quantitative measures currently available. Some of the quantitative observations offered by Brooks are not inconsistent with the rankings shown in Table I for system quality.

**EVALUATION**

A review of the evidence indicates a strong leaning in one direction as the Overall summarizes in Table I. The availability of this evidence gives the opportunity to take a fresh look at some alternatives and make rational choices to fit given situations and needs. Let us consider local, internal, and external data, for example.

Local variables are items of data made available in the design and implementation to only selected modules. If they are accessible only in the modules in which they are declared or defined, then they are true internal variables. If they are accessible to other modules, then all of those modules must be in the same branch of the tree that represents the nested hierarchical relationship among the functions. Effectively, the locals thus used become "restricted" globals. Local variables are well regarded in the theory of structured programing.

Used with global data, local data are a step in the direction of passed data. Used with passed data, local data offer a relaxation of the visibility enforced with passed data. But the use of true internals degrades neither passed data nor global data usage. Internal data use may even enhance either but especially global data.

"External" data are another possibility. These are items of data communicated between specific modules anywhere in the hierarchy—not just between immediate superiors and subordinates. It is like a private line for communication external to and addition to those provided by the passing of data.

External data are better regarded in structured design than global data because they are more visible to people. But their flow does not follow the tree, and as they are usually
602 National Computer Conference, 1978

TABLE I.—The Most Favorable Rank is 1, the Least Favorable is 4.

<table>
<thead>
<tr>
<th>Evidence</th>
<th>Passed Data</th>
<th>Global Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Factor</td>
<td>Leveled</td>
<td>String</td>
</tr>
<tr>
<td>Volume</td>
<td>2 1</td>
<td>4 3</td>
</tr>
<tr>
<td>Values</td>
<td>4 3</td>
<td>2 1</td>
</tr>
<tr>
<td>Congruence</td>
<td>2 1</td>
<td>3 5 3.5</td>
</tr>
<tr>
<td>Coupling</td>
<td>2 1</td>
<td>3 5 3.5</td>
</tr>
<tr>
<td>Strength</td>
<td>2.5 2.5</td>
<td>2.5 2.5</td>
</tr>
<tr>
<td>Overhead</td>
<td>4 3</td>
<td>2 1</td>
</tr>
<tr>
<td>Convenience</td>
<td>2 1</td>
<td>4 3</td>
</tr>
<tr>
<td>Maintainability</td>
<td>2 1</td>
<td>4 3</td>
</tr>
<tr>
<td>Quality</td>
<td>2 1</td>
<td>3.5 3.5</td>
</tr>
<tr>
<td>OVERALL</td>
<td>2 1</td>
<td>4 3</td>
</tr>
</tbody>
</table>

used, they provide another (sometimes complicating) data flow for people to understand simultaneously with either passed or global data. A common use of external data accessibility is the use of scratch files to communicate data between selected modules, often relatively high in the tree.

Is there some form of intrasystem data communication that would rate better than any of those discussed thus far? Consider the following possibility, termed "monitored data," that combines the strong features of the forms and varieties already discussed.

1. Define or declare with a unique name each and every data item only once in the software system to be created. Incorporate in the definitions or declarations any list-linking or component or group-element relationships among the items of data.

2. Enable by a specific visible means in the documentation for the module, each module that may access the value of any given item of data or may assign the value, on an item by item basis, noting the role of the item of data. The roles may be for control, for processing, for modification of value, or for communication through unchanged.

3. Record and associate with the definition or declaration, the identification of the accessing and assigning modules and the roles of the data in those modules. This documentation, while not intrinsic to monitored data, adds substantially to the human convenience and to the maintainability.

4. Disnable all other modules for all items of data by a means that prevents their implementation if any access is explicitly or implicitly attempted in or by them to any item of data for which they have not been specifically enabled.

5. Make any access to any item of data be to all parts of that item of data (for example, enabling for a 1000-byte record yet only using a 2 byte field in it would be illegal) except where explicit decisions result in alternative access.

Some comparison of monitored data with the forms and varieties considered earlier may be interesting (see Table I).  For systems and programs involving more than about 50 intermodule items of data, on volume, congruence, coupling, and strength, the monitored data alternative ranks the same as the passed-string alternative. On values and overhead, it ranks the same as the global-string alternative. On system quality, it ranks better than the passed-string alternative, by close to a factor of two. For systems and programs of smaller size and for some strongly mathematics oriented systems, the passed-string alternatives appear to be superior to the use of monitored data. But from the limited experience thus far, for most non-trivial systems and programs implemented in COBOL, FORTRAN, PL/1, JOVIAL, and BASIC, the monitored data appear to be the superior alternative.

Insufficient experience is available to give a confident ranking on convenience and maintainability. Some observations may be indicative, however. On convenience, the monitored data appear to offer all the advantages of both the passed and the global alternatives, and avoid the low-visibility of the global and some of the redundancy of the passed data alternatives. The verification of the enabling in points # 2 and # 4 requires either a manual or a software step. An automatic software step improves the human convenience. The usefulness and precedent for such steps already exists in some structured programming practices using precompilers, data base administrators, librarians, and project files. 3 Input-output tables can also be a useful aid. 7 Because of the excellent visibility it produces, the monitored data alternative appears to offer a maintainability potential superior to that of the passed-string alternative.

CONCLUSION

Among the four alternatives considered, the weight of the evidence favors the passed form of intrasystem data communication, especially where the overhead is not of serious concern. The disjoint string ranked superior or equal to the use of data in levels of abstraction on all nine factors considered. Except on convenience and maintainability where little evidence exists yet, the monitored form of intrasystem data communication appeared to be superior or equal to the passed-string alternative.

REFERENCES
