Journal of Official Statistics
Vol 4. No. 4, 1988, pp. 333–348

CATI Instrument Logical Structures: An Analysis With Applications

Matthew Futterman[1]

Abstract: This paper identifies and examines the logical relationships commonly found in computer-assisted telephone interviewing (CATI) instruments and describes a method of programming a computer to recognize these relationships as logical structures. CATI systems can use instrument logic databases to generate instrument flow charts, display instrument logic at interviewing stations, and augment or replace keyboard-based interview movement commands with mouse-based commands. The paper includes a discussion of how the properties of logical structures can be applied to give the instrument designer more control over the instrument's content and function.

Key words: CATI; computer-assisted telephone interviewing; CATI instrument; logical structure; adjacency matrix; one-entry/one-exit structure; database.

0. Introduction

At a conference on computer-assisted telephone interviewing (CATI) a few years ago, one of the participants asked if it was possible to display "you-are-here" diagrams to CATI interviewers as a way of guiding them through complex interviews. I thought this was an intriguing idea. We use flow charts, after all, to help us design computer programs, why not use similar diagrams to help us run them?

The idea became a springboard for other ideas. If we are able to show interviewers a diagram of an instrument, can we also have them use the diagram to "navigate" their way through complex logic? And can this diagram have built into it some way of checking to see that they are navigating correctly? Perhaps we can have the computer create printouts of these diagrams for us to use as diagnostic tools during the instrument design cycle. Or maybe we can have the computer scan a printed diagram in order to record the logic it represents.

This paper proposes a way for CATI systems to accomplish these kinds of tasks. I begin by analyzing CATI instrument structure. What logical relationships do we find in CATI instruments? How can these relationships be graphically represented? Next, I at-

[1] Head of CATI system development for the Institute for Social Science Research at the University of California, Los Angeles, CA 90024, U.S.A.

tempt to show how a computer can be programmed to recognize logical structures and transform them into data. Then I suggest ways CATI systems can use the data to perform diagnostic and run-time functions. The paper concludes with a discussion of how the properties of logical structures can be applied to give the instrument designer more control over instrument function and content.

1. Visually Representing CATI Instruments – Some Benefits

Computer scientists tell us that we should plan our design of programs so that they consist of small (on the order of one page), highly functional subprograms, since small programs are easier to understand than large ones (McGowan and Kelly (1976)). This is the underlying philosophy of modular design.

Interestingly, CATI instruments fit well with the modular design model, since the instrument item is the ideal module; it is small and contains highly specific, highly functional logic. When represented on the printed page, the purpose and function of an item is readily apparent. We can list all the items in an instrument, examine each one individually, and assure ourselves that we pretty much know what each of the items is supposed to do.

Problems arise when we try to synthesize these individual functions in order to determine their effects upon, and interactions with, one another. So throughout the instrument design phase we draw diagrams and charts that represent the manner in which we want the instrument to behave. Can a computer be programmed to draw these diagrams for us?

In order to draw a diagram of a CATI instrument, or, for that matter, any computer program, we (humans or computers) must have a detailed understanding of the logical relationships among its components. It is convenient to think of such a diagram as an illustrated cross-reference. Computer programs that are used as tools by programmers, such as linkers and compilers, have traditionally created cross-references. What should a CATI compiler attempt to cross-reference for us, and how should this information be represented diagrammatically?

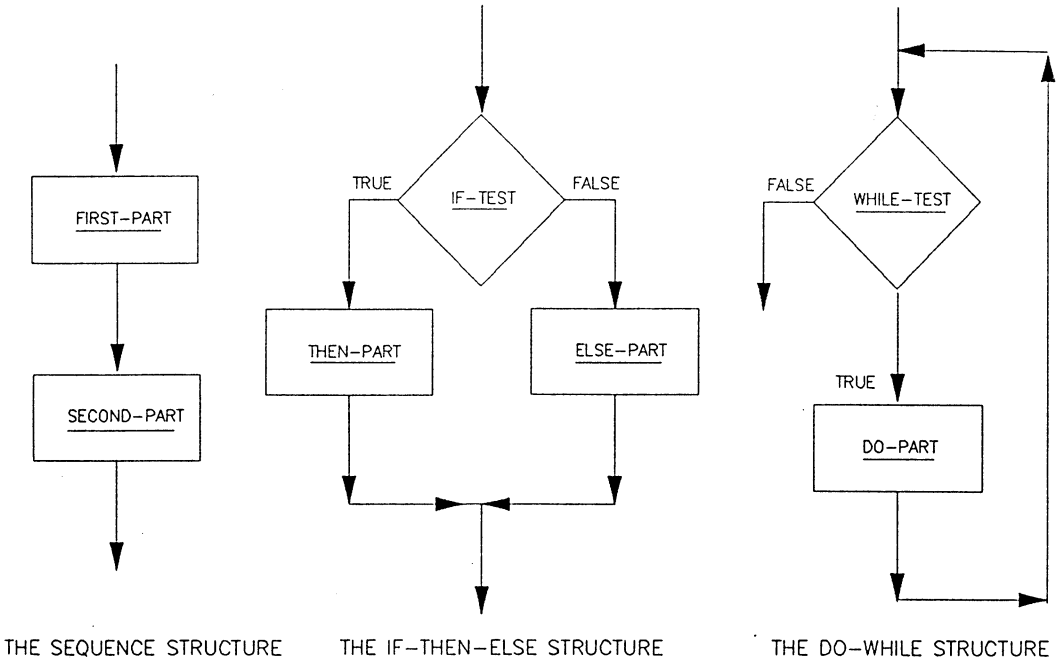
Computer scientists have identified three "canonical" structures that form the basis for all structured programs (Fig. 1). If we were to draw, then, a diagram of a structured program, a good starting point would be to identify its component canonical structures and draw them, in order, from beginning to end. By including all the interconnections among the structures as well as a unique name for each, we would then have what might reasonably be considered a good visual representation of the structure of the program.

We can accomplish the same thing for a CATI instrument – that is, the graphical representation of its logic – by identifying the component structures of instruments in general, and by programming the CATI compiler to both recognize these general relationships and to identify the structural relationships in individual instruments. Armed with such a tool, we could then have the computer graphically represent the logical structure of our instrument at any time during the design phase in the same way we now have it create conventional listings. Jabine (1985) and House (1985) have both discussed ways in which structural diagrams can be beneficial during the instrument design cycle.

There would be other benefits as well. A CATI compiler programmed to recognize and cross-reference instrument structural information has the potential of relieving the CATI instrument designer of much of the burden of defensive programming, that is, programming to protect instrument logic from random alterations during an interview.

We would also have the ability to show interviewers where they are in an interview,

Fig. 1. The three canonical structures adapted from McGowan and Kelly (1975)



how they got there, and where they are likely to go next. We would even be able to use a mouse instead of a keyboard to move to different parts of an instrument.

2. The Logical Structure of CATI Instruments

2.1. The basic components

What does a CATI instrument look like?

Perhaps the way most of us envision a CATI instrument logical structure is as a flow chart. Flow charts have traditionally been used to represent the logical flow of computer programs and they are useful tools for developing and understanding survey questionnaires, including CATI instruments (Jabine (1985)).

Graphs are another way to represent logical structures. Directed graphs (digraphs), routing graphs and subgraphs are examples of types of graphs that may be familiar to some readers. Gondran and Minoux (1984) and Knuth (1973) are good sources of general graph theory. Willenborg (1987) places graph theory in the specific context of CATI instrument design, particularly with regard to the issues of complexity and balance.

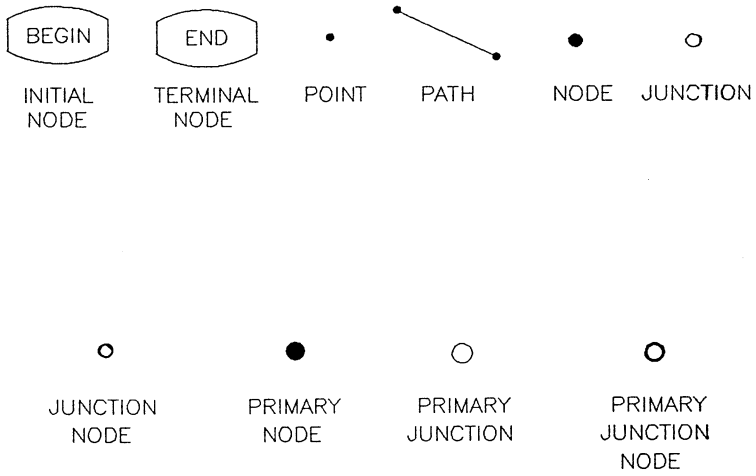
This paper describes logical structures in a manner that is similar to the way they are described by flow charts and graph theory. An attempt has been made to define terms so that they are familiar to those who design CATI instruments. Readers familiar with graph theory may wish to consult Table 1 for an informal cross reference of the different terminologies. All readers should refer to Fig. 2, which shows the symbols that are used here to represent the basic structural components.

Table 1. Terms used in this paper versus terms used in graph theory

Terms used in this paper	Corresponding terms used in graph theory
complete path	full-length path
initial node	source, initial endpoint, initial vertex
mandatory point	cut-point
path	edge, oriented path, arc, path
point	vertex, node, point
section	sub-digraph, subgraph
supersection	directed graph, digraph, routing graph
terminal node	sink, final vertex, terminal endpoint

Source: Gondran and Minoux (1984); Knuth (1973); Willenborg (1987).

Fig. 2. Basic components of logical structures



How, then, shall we represent CATI instrument logic?

Let us start with how we wish to represent an item. We define an ITEM as having a unique name with respect to all other instrument items, and a branching specification, that is, an instruction to the computer that tells it which item to process next. Items may have other attributes as well, but we shall consider these to be optional. We will use a POINT and a label to represent an item and its name, respectively, as in Fig. 3.

We shall represent branching instructions as lines that connect the items according to their logical relationships, with the implied logical flow proceeding from top to bottom. For example, to indicate that the computer is to process Q2 immediately after processing Q1, we draw a line from Q1 to Q2, with Q1 placed above Q2 in the diagram, as we have in Fig. 3. We shall refer to this line and the logical relationship it represents as a PATH.

Next, let us represent a very simple instrument consisting of Q1, Q2, a point of entry into the instrument labelled BEGIN, and a

Fig. 3. A supersection

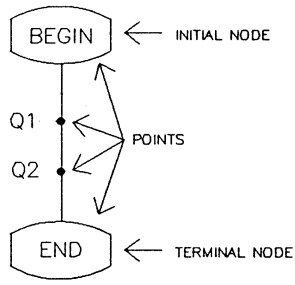
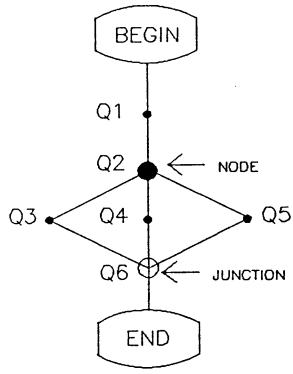


Fig. 4. A supersection with a node and a junction



point of exit out of the instrument labelled END. The point of entry is the INITIAL NODE, the point of exit the TERMINAL NODE. We shall define as a SUPERSECTION an instrument that has an initial node, a terminal node and any number of points in between.

More complex branching instructions, specifically branching instructions that are conditional rather than unconditional, are represented by drawing a path for each condition. Thus if Q2 is an item that branches conditionally to either Q3, Q4, or Q5, we draw a path from Q2 to each of these items, as in Fig. 4. An item such as Q2 can be thought of as a switch; it selects which path to follow next. We shall call an item that functions as a switch a NODE. Only one path can be selected by a node each time it is executed by the computer, and that path then becomes part of the ACTIVE PATH. A node can be executed again at a later time under different conditions that cause a new path to be selected, thus altering the active path. By contrast, a point that is not a node can never alter the active path.

Now assume Q3, Q4, and Q5 all branch unconditionally to Q6. Observe two things about this configuration, shown in Fig. 4. First, there are three paths leading to Q6, the paths from Q3, Q4, and Q5. A point where two or more paths converge, as they do at Q6, is called a JUNCTION. Second, there are no paths interconnecting Q3, Q4, and Q5. Thus if the computer is executing Q3, for instance, there is nothing in the instrument's logic that will cause the computer to execute Q4 or Q5; only the occurrence of an external event, such as an interviewer command, can cause Q4 or Q5 to be executed.

Before we get involved in a discussion of more complex structures, let us observe some of the important properties of our example supersection. Notice that we can trace three distinct paths that lead from the initial node to the terminal node. These are each COMPLETE PATHs. Only one of these paths will actually have been taken, however, when we finally reach the terminal node (as when an interviewer completes an interview) because of the switching properties of nodes. The complete path that is actually taken is called

Fig. 5. A complete path

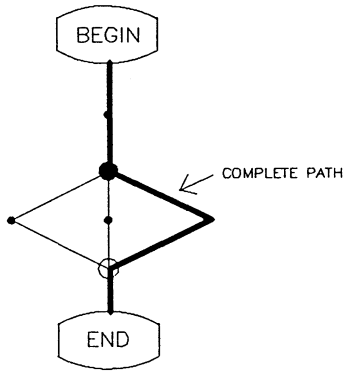


Fig. 6. Mandatory paths

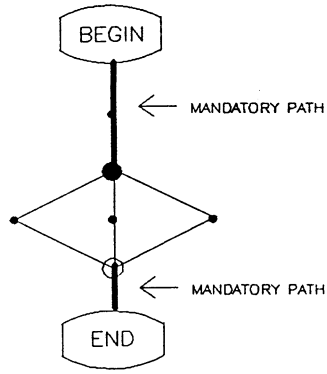
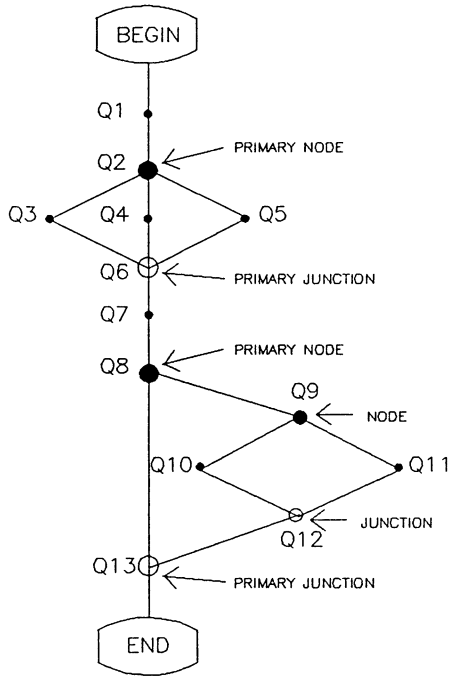


Fig. 7. A supersection with primary nodes and primary junctions



the ACTIVE COMPLETE PATH (Fig. 5). Knowing which points lie on the active complete path is important, since it tells us not only which items were executed, but also which item responses are applicable. A stored response is applicable only if the item that stored it is on the active path.

Another important thing to note is that some points are common to all complete paths. These points, called MANDATORY POINTs, tell us which items will always be executed. Similarly, MANDATORY PATHs, paths which are common to all complete paths, identify sequences of items that are always executed. See, for example, Fig. 6. Keep in mind, however, that it is possible to construct an instrument that has only two mandatory points, the initial and terminal nodes.

2.2. Decomposing large structures into smaller structures

Now that we have identified some of the key properties of a supersection, let us look at an example that is more complex and see how it can be decomposed into smaller structures.

To start, let us add some additional items, Q7 through Q13, to our sample instrument, shown in Fig. 7, in order to increase its complexity, and make a list of all the nodes, junctions and mandatory points in the supersection. Are any of the items in more than one category? Items BEGIN, Q2, Q8, and END are nodes that are also mandatory points. We shall classify them as PRIMARY NODEs. Similarly, items Q6 and Q13 are junctions that are also mandatory points, and we will refer to them as PRIMARY JUNCTIONs. It is also possible for JUNCTION NODEs and PRIMARY JUNCTION NODEs to exist, though there are none in this example.

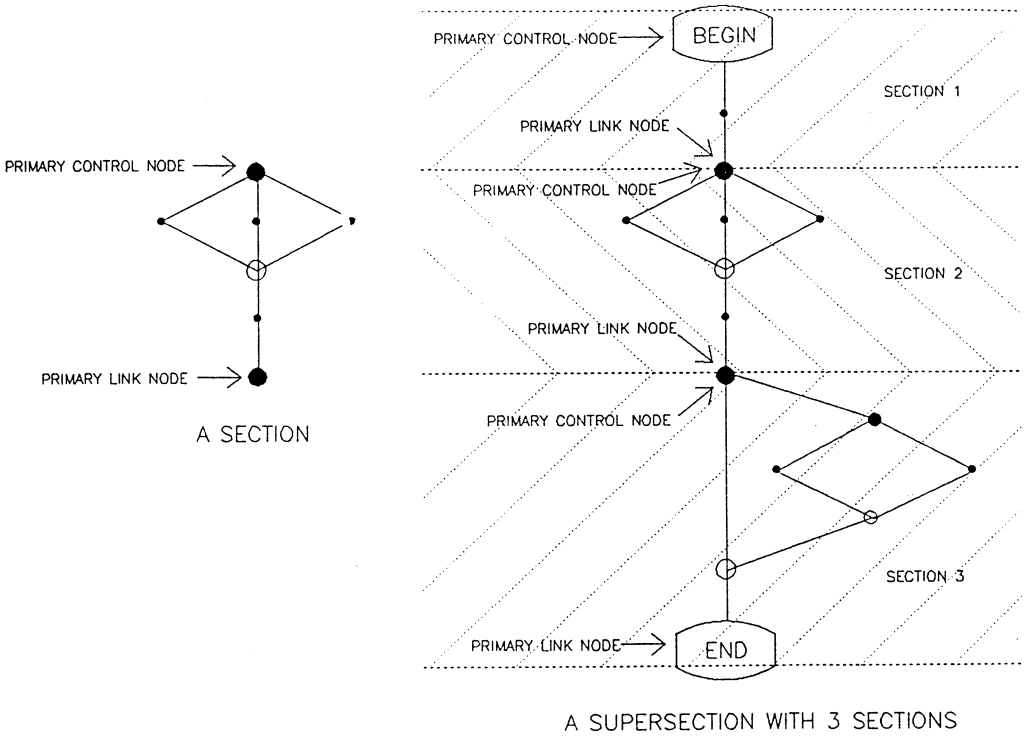
Recall that nodes have a switching function that allows them to select portions of the active path. Primary nodes always select portions of the active complete path, which suggests that they play an important part in the makeup of an instrument's structure. In fact, identifying the primary nodes allows us to decompose a complex supersection into smaller structures called SECTIONs, which we shall define as consisting of all the points bounded by and including adjacent primary nodes. See Fig. 8 for an example.

Sections have interesting parallels to supersections. For instance, the primary node that begins a section, which we shall refer to as a PRIMARY CONTROL NODE, is similar to the supersection's initial node; each SECTION PATH begins with the primary control node. Similarly, the primary node that ends the section, which we call the PRIMARY LINK NODE, is similar to the supersection's terminal node, since all section paths terminate there. The one section path that is selected as we proceed from the primary control node to the primary link node is called the ACTIVE SECTION PATH. If it extends all the way to the primary link node then it is the ACTIVE COMPLETE SECTION PATH. The portion of the mandatory path that passes through a section is called the MANDATORY SECTION PATH.

Because of the similarities to supersections, sections provide us with a way of comprehending smaller portions of the logic of a complicated instrument. And, like items, sections can be viewed as instrument building blocks, building blocks that are larger and more comprehensive than items. Let us take a closer look at sections and identify their properties and component structures.

Every section has exactly one primary junction, that is, one point where all section paths converge. If the last point of a section – its primary link node – is a junction, then it is a primary junction. Otherwise, the junction immediately preceding it is a primary junction. Why are primary junctions important?

Fig. 8. Decomposing a supersection into sections



They help us to determine portions of the mandatory path. It happens that every point between a primary junction and a primary link node is a mandatory point. Also, there are never any mandatory points between a section's primary control node (its first point), and its primary junction. Thus a section that has a primary junction link node has only two mandatory points, the primary control node and the primary link node.[2]

2.3. *Structural complexity and path analysis*

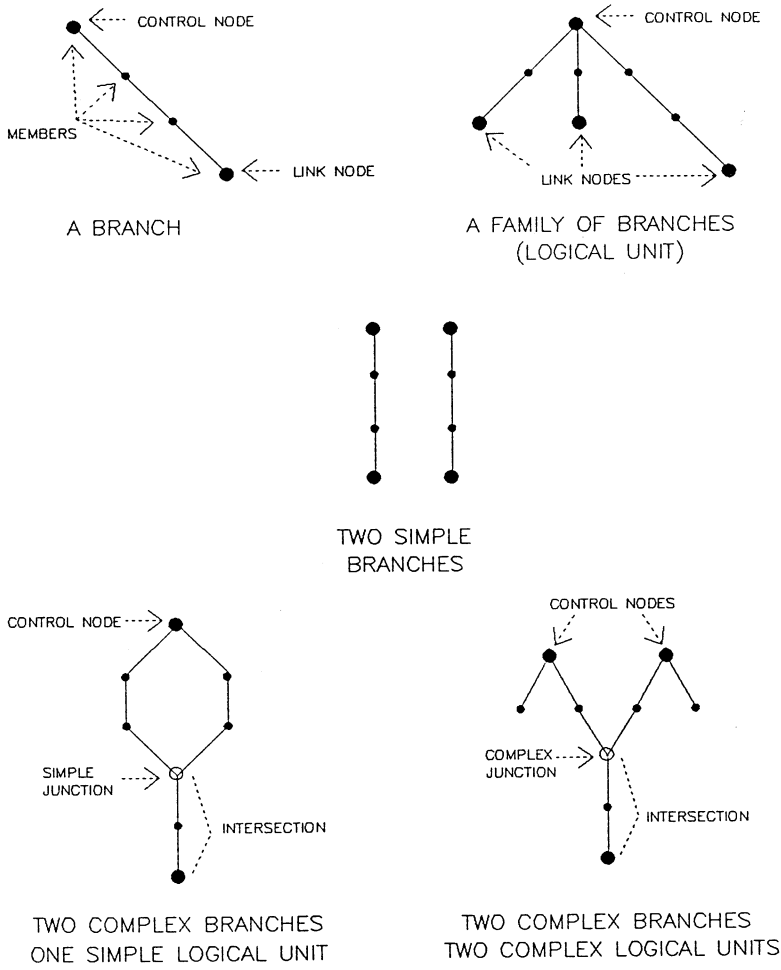
Thus far we have discussed points that lie on the mandatory path, particularly primary nodes and junctions. What significance do the secondary nodes and junctions have?

We can think of the primary nodes as shaping the structural outline of a supersection, whereas secondary nodes fill in the details by combining to form smaller structures. These smaller structures, which we shall now examine, include branches and logical units (Fig. 9).

The portion of a path between two adjacent nodes is called a BRANCH. The top node is the CONTROL NODE, the bottom node is the LINK NODE. There can be any number of points between the two nodes. Each point on the branch is called a MEMBER of the branch. The set of branches emanating from a particular control node is called a FAMILY of branches; the set of all the members of a family is a LOGICAL UNIT. Each time a control node is executed, it can select from its family exactly one branch, which then becomes the ACTIVE BRANCH.

[2] The one exception is the special case of a section that begins with an initial node that is not a logical node, that is, an initial node that does not branch conditionally.

Fig. 9. Simple and complex structures



Junctions contribute to the complexity of a logical structure. For example, a SIMPLE BRANCH is a branch whose members, excluding the control and link nodes, are not members of any other branch. A COMPLEX BRANCH, on the other hand, has at least one non-node member that is also a member of another branch. The set of members that are common to another branch is called an INTERSECTION. The top-most member of an intersection is always a junction.

Similarly, a logical unit is a SIMPLE LOGICAL UNIT if none of its non-node members are members of any other logical units, whereas a COMPLEX LOGICAL UNIT has at least one non-node member that is also a member of another logical unit, with a junction as the point of intersection.

Why make this distinction between simple and complex structures? Consider the situation where the computer is executing a non-node member of a simple branch. Once we determine which item is the control node for the current item, we can identify, with certaintly, a portion of the active path. In this case, that is the path between and including

the control node and the current item. Note that there is no need to test the logic at the control node to make this determination.

But what if there is a junction between the current item and the control node? Now we are dealing with a complex branch. The control node must be tested in order to determine which path was taken to the junction.

The situation gets more complicated if the junction is a member of two or more logical units (a COMPLEX JUNCTION), since now there are at least two control nodes that share responsibility for choosing a common portion of the path (intersection). Which control node do we test in order to determine the path? It is not possible to answer that question without knowing which control node is on the active path. How far back up the structure will we have to go in order to make that determination? Until we find a node that is on the active path.

There is another way to approach the problem, however. We know, for instance, that primary nodes are on the mandatory path. We can deduce that if a particular primary node has been executed, then it is on the active path. Furthermore, since every point is part of a section, if a particular point is being executed, then there is a primary control node on the active path preceding that point. The active path can then be traced by testing the primary control node, which will then lead to the next active node, which is then tested, and so on, until we arrive at the current point. In other words, in order to locate any currently executing point, we merely need to trace the active section path.

Similarly, in order to determine whether a nonexecuting point is on the active path, first determine which section it belongs to and then check whether its primary control node is on the active path. If it is not, then neither is the point in question. If it is, then trace the active section path. If the primary link node is reached without having passed through the point in question, then it is not on the active path.

2.4. *One-entry/one-exit structures*

Here we have seen some examples of how a section can be used to focus on part of a larger structure, thus giving us some valuable insight into the details of smaller structures. What is it about sections that makes them so useful?

Perhaps the most important property of a section is that there is a single point of entry to, and a single point of exit from, the structure. This type of structure is called a ONE-ENTRY/ONE-EXIT structure and is important to the concept of structured programming. Large, complex logical structures – CATI instruments or computer programs in general – are easier to design, modify, document, and maintain if they are composed entirely of smaller, one-entry/one-exit structures (McGowan and Kelly (1976) and House (1985)). In fact, the three canonical structures that form the basis of structured programming are each structures of this type.

So far we have discussed several one-entry/one-exit structures including supersections, sections, and simple branches. There is another one – a "section within a section" – called a SUBSECTION. A structure is a subsection if it meets both of the following conditions: (1) – all paths that emanate from a secondary control node i converge at a secondary link node j; (2) – all paths that converge at a secondary link node j emanate from a secondary control node i. The two nodes that begin and end a subsection are called a SUBSECTION CONTROL NODE and a SUBSECTION LINK NODE, respectively.

There are many similarities between sections and subsections, primarily because they are both one-entry/one-exit structures. This makes the internal logic of subsections, like

that of sections, relatively easy to comprehend. For instance, if we know that a subsection control node is on the active path, we can readily trace the ACTIVE SUBSECTION PATH. We know also that all subsection paths converge at the subsection link node.

Thus far we have assumed that all branching proceeds from top to bottom, that is, in the direction of the terminal node. This is consistent with good structured programming techniques. However, structured programming allows for one exception, that of conditional iteration, which is the ability to execute some portion of a program repeatedly until a condition goes false. To allow for this capability, we will define a structure called a ONE-ENTRY/ONE-EXIT LOOP, which has exactly one point of entry into the loop and exactly one point of exit out of the loop. Confining conditional iteration to a one-entry/one-exit structure permits us to retain our assumptions about the other logical structures.

2.5. *Pseudonodes*

Our discussion has focused thus far on LOGICALLY BOUNDED structures, structures that begin and end with nodes and junctions. A computer can be programmed to recognize logically bounded structures without needing to know anything other than structural logic, since logical entities – nodes and junctions – determine where one structure ends and another one begins. (See Table 2.) What if we want to define structures that are not strictly determined by logic, but instead are based more on content or function?

In order for a computer to recognize such structures we have to give it some information in addition to logic. One way to do this is to create a type of item called a PSEUDONODE. A pseudonode is simply an item that we declare to be a node in the same way we declare a storage location to be an integer; we give a computer an instruction to treat an object as a particular type of object. Declaring items as pseudonodes gives us more control over structure, since any item we declare as a pseudonode will be considered as a structural boundary just as if it were a node.

Consider two examples from Figure 7. First, by declaring mandatory point Q1 as a pseudonode, we form a new section bounded by Q1 and Q2. Note that Q1 is now a primary node since it is a (pseudo)node and a mandatory point. Second, by declaring junction Q12 as a pseudonode, we form a subsection bounded by Q9 and Q12. Q12 is now a junction (pseudo)node.

In the discussion that follows, we discuss how a computer can be programmed to recognize structures and how the structures can be used in applications and instrument design.

Table 2. Entrances to and exits from logically bounded one-entry/one-exit structures

Structure type	Entrance	Exit
Supersection	Initial node	Terminal node
Section	Primary control node	Primary link node
Subsection	Control node	Link node
Simple branch	Control node	Link node
Mandatory path	Primary junction	Primary node
Loop	Junction	Node

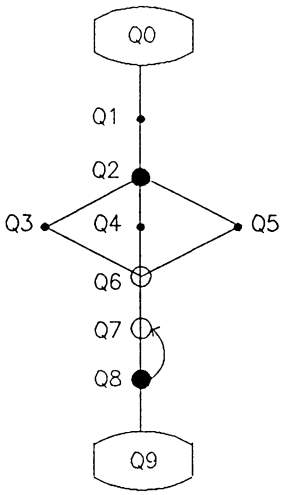
3. A Database of Logical Structures: Some Applications

3.1. Constructing a database

We have now identified several types of structural components – nodes, junctions, branches, to name a few. How might we program a computer to recognize them?

Graph theory gives us an important tool called an adjacency matrix, which describes logical relationships in tabular form. Assume we have a supersection with n items numbered top to bottom from 1 to n and that, except for the case of a one-entry/one-exit loop, an item always branches to an item or items with a higher number. We then construct an n by n table. For each item i that has a path to an item j we place a 1 in cell ij, otherwise cell ij has a 0. We can use the table to determine whether there is a path between any two items i and j by examining the value in cell ij. Further, if row i has two or more 1's, then item i is a node; If column j has two or more 1's, item j is a junction. If row i has a 1 in a cell ij that is in the lower left triangle of the matrix, then item i loops (branches backwards) to item j; If column j has a 1 in a cell ij that is in the lower left triangle of the matrix, then item j is a loop (backward branch) target of item i. Fig. 10 shows a supersection and its corresponding adjacency matrix. Note that the loop from Q8 to Q7 produces the only 1 in the lower left triangle of the adjacency matrix.

Fig. 10. Using an adjacency matrix to describe the logic of a supersection

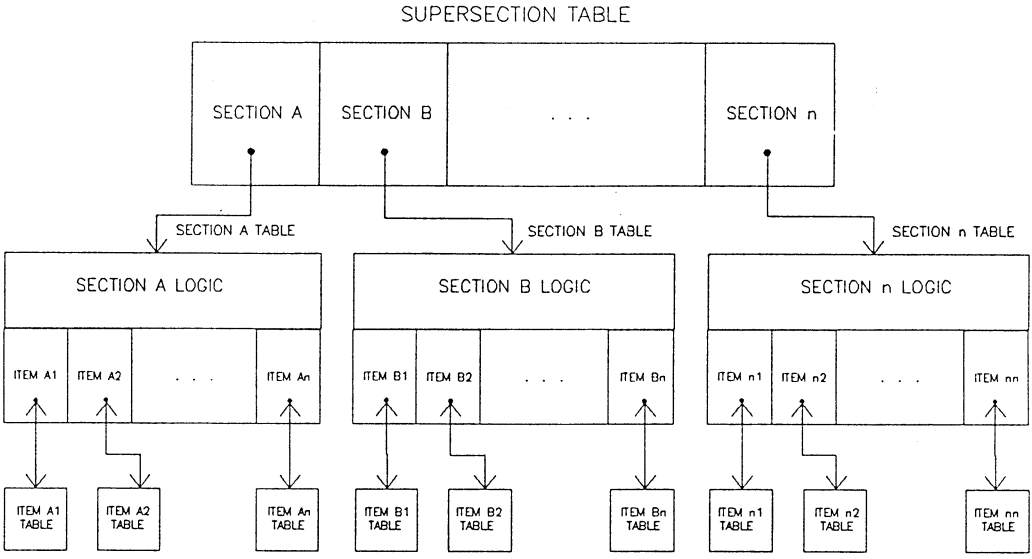


SUPERSECTION

	Q0	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
Q0	0	1	0	0	0	0	0	0	0	0
Q1	0	0	1	0	0	0	0	0	0	0
Q2	0	0	0	1	1	1	0	0	0	0
Q3	0	0	0	0	0	0	1	0	0	0
Q4	0	0	0	0	0	0	1	0	0	0
Q5	0	0	0	0	0	0	1	0	0	0
Q6	0	0	0	0	0	0	0	1	0	0
Q7	0	0	0	0	0	0	0	0	1	0
Q8	0	0	0	0	0	0	0	1	0	1
Q9	0	0	0	0	0	0	0	0	0	0

ADJACENCY MATRIX

Fig. 11. A database of logical structures



A computer can readily be programmed to construct an adjacency matrix by examining the branching instructions of each item and recording the results in a table. By devising the appropriate algorithms, the computer can also be programmed to identify the more complex logical relationships such as logical units and sections. It is beyond the scope of this article to discuss these algorithms in detail. However, a key step would be to identify each of the primary nodes (including primary pseudonodes), since this quickly yields each of the sections and their boundaries.

Once the computer is programmed to recognize logical structures, the next step would be to construct a hierarchical database in order to record and cross-reference the relationships. How would we want to organize such a database? We have seen that sections describe discrete portions of the logic of supersections and are in turn composed of smaller portions, including items. A hypothetical database then might consist of a supersection table that has references to section tables, one for each of its component sections. The section tables, in turn, would contain references to each of its component items, as well as information that summarizes the logical relationships among the items, including logical units, subsections, branches, nodes, junctions, and so on. At the lowest level, each item table, in addition to the usual information such as branching instructions and text, would contain a reference to its section. Fig. 11 illustrates this database organization.

3.2. *New tools for CATI*

The computer can use a database of logical structures in various situations in order to perform different types of functions. For example, a CATI compiler can use the database to create conventional and graphical cross-references to aid the CATI instrument designer during the development and testing of an instrument. A CATI run-time module – the program that administers the on-line CATI instrument – can use the database to ensure

that nonstandard instrument movement requests (interviewer jump commands) are allowed only if consistent with the instrument's internal logic given the responses stored for a particular interview.

The run-time capabilities extend even further, however, because the structural representation of the instrument can be displayed at the user's (interviewers, supervisors, project managers) screen. The ability to display an instrument to interviewers in this manner during the training stage and during actual interviews can be beneficial (Jabine (1985)).

If we combine the ability to have the computer ensure nonstandard movement correctness with the ability to display instrument logical structure, the next step would be to have interviewers use a mouse to "navigate" their way through an instrument. This would free interviewers from concerns about which interview movement commands to use and when to use them. Concerns about proper command syntax would be a thing of the past as well.

Yet another application would be to enter an instrument's logical structure into the computer by using a scanning device to digitize a structural diagram. This amounts to reversing the process we have described above; instead of first using the CATI compiler to program an instrument and then have it draw us a diagram of the resulting logical structure, we would first provide the compiler with a finished structural diagram in order to have it create a computerized template of the corresponding instrument logic. The benefits of doing this could be quite substantial.

For example, the CATI systems of today have many instances of incompatibility with one another, including different instrument programming languages, differences in compiler (or interpreter or translater) design, different database formats, different file formats and so on. The list is quite large. It may not be realistic to expect widespread standardization in all of these areas at this late date. However, it is not too late to consider standardizing the way we wish to graphically represent instrument logical structure. Doing so would allow instrument logic to be seamlessly transported among otherwise dissimilar CATI systems using a standardized graphical medium and having the ability to recognize logical structures.

3.3. *Managing instrument content*

So far this discussion has focused strictly on logical structure. We have made no attempt to relate instrument logic to instrument content. Is there a relationship, and if so, how can principles of logical structure be applied to content structure?

An instrument can be designed so that two distinct topics – let us say respondent medical history and respondent demographics – are presented in sequential fashion, for instance, all of the items that ask about medical history precede all of the items that ask about demographics. At the other extreme, the two topics can be highly interleaved. Each arrangement will have its own distinct logical structure. Yet can we infer from a diagram of either of these logical structures anything about its content?

The answer is no. Any association that we wish to establish between an instrument's logic and its content must be carefully planned during the instrument design process. Thus if we want the computer to recognize the medical history items as being distinct from the demographic items, we must place the two types of items in separate logical structures. One solution is to design an instrument that has a supersection with two (logical) sections, one for the medical history items, the other for the demographic items. With this scheme, we can use the logical structure database to access and manage the instrument content structure.

For example, an interviewer using a mouse could move to contextually distinct portions

of an interview by clicking on a descriptively labelled part of a logical structure. Selections would be among informatively labelled topics such as "medical history," "tobacco use," and "household information," rather than among terse, less informative item names like "M0," "T21," and "HCNT."

As another example, we could include in the section table database such attributes as OPTIONAL and MANDATORY. A section with the MANDATORY attribute would require that the section have a complete section path in order for an interview to be considered complete (the default case). A section with the OPTIONAL attribute would not have this requirement. Thus, we could design a hypothetical instrument in which we include a mandatory "household information" section and an optional "tobacco use" section.

By designing instruments that have one-to-one correspondences among logical structures and content structures, we should find it possible to manage instrument content in other ways as well.

3.4. Managing instrument functions

CATI instruments, in addition to their content, also consist of items that perform identifiable functions such as "calculate respondent's net income," "display interview break-off sequence," and "obtain next call appointment time." Like content structures, functional structures must be associated with logical structures during the instrument design process in order for us to use logical structures to control their access and execution. How effectively we control access and execution is an important aspect of instrument design.

Often, for example, a group of items combine to perform a single function. For instance, in order to calculate a respondent's net income, one item might be responsible for initializing the storage locations that are to be used as part of the calculation, another item or group of items might be used to ask the respondent his or her gross income, others to obtain the amount of taxable income, and still others to perform the calculations. It is essential that each of these items be executed in the correct sequence in order to perform the "calculate respondent's net income" function correctly. How can we guarantee that they are?

Nicholls and House (1987) have proposed the concept of a VIRTUAL ITEM, a structure which may include one or more items "but which functions collectively like a single item." The key requirement of a virtual item is that it must be accessible only through its first component item.[3] Thus a virtual item is a one-entry (though not necessarily a one-exit) functional structure and is similar to several of the logical structures we have discussed, including sections and subsections. By allowing access to the structure only through its first item, we can guarantee that the component items will always be executed in their proper order. How can we restrict access to a structure in this way?

The key to controlling access to a functional structure is to clearly establish its relationship to a corresponding logical structure during the instrument design process – the same strategy described above to manage content structures. Then we can utilize the properties of one-entry/one-exit structures such as sections and subsections to control access to and execution of a functional structure.

For instance, we can include in the section table database the attributes PROTECTED and UNPROTECTED. A section marked as

[3] Nicholls and House suggest one other restriction: that only the first item of a virtual item be a displayed item (an item with displayable text). This precludes interviewers from accessing subsequent items by using interview movement commands. The implementation of the virtual item concept as described here avoids this restriction.

UNPROTECTED allows access to any item in the section as long as it is on the active path; a section marked PROTECTED denies access to any item in the section except the primary control node.

Let us see how this can be useful. Suppose we wish to implement the "calculate respondent's net income" function described above. By including all of the items associated with this function – a virtual item – in a section, we could then declare the section to be PROTECTED. During an interview, all interviewer requests to jump to items within the section – except for requests to jump to the primary control node – are denied, either by rejecting the request completely or by routing the request to the primary control node. In this way we can quarantee that the items that comprise the function are always executed in the correct sequence. Further, if we declare the section to be MANDATORY, we can guarantee that the function will have to be fully executed in order to obtain a complete interview.

I have suggested only some of the ways to manage instrument structures. The section table database can be extended to include other types of control attributes and status indicators. Subsections have the potential of giving us intricate control of CATI instrument functions and content. Subsections, as well as other one-entry/one-exit structures, can be implemented as subroutines. Others may wish to explore these possibilities more fully.

4. References

Gondran, M. and Minoux, M. (1984): Graphs and Algorithms. John Wiley and Sons, New York.

House, C.C. (1985): Questionnaire Design With Computer-Assisted Telephone Interviewing. Journal of Official Statistics, 1, (2), pp. 209–219.

Jabine, T.B. (1985): Flow Charts: A Tool for Developing and Understanding Survey Questionnaires. Journal of Official Statistics, 1, (2), pp. 189–207.

Knuth, D.E. (1973): The Art of Computer Programming – Vol. 1, Fundamental Algorithms, Addison-Wesley, Reading, MA, pp. 362–406.

McGowan, C.L. and Kelly, J.R. (1976): Top-Down Structured Programming Techniques. Petrocelli/Charter, New York.

Nicholls, W.L., II, and House, C.C. (1987): Designing Questionnaires for Computer-Assisted Interviewing: A Focus on Program Correctness. Proceedings of the Third Annual Research Conference, U.S. Bureau of the Census, pp. 95–111.

Willenborg, L.C.R.J. (1987): The Routing Structure of Questionnaires. In CBS Select 4-Automation and Survey Processing, Centraal Bureau Voor de Statistiek, Voorburg, Netherlands, pp. 97–106.

Received December 1987
Revised January 1989