THE CASE DESCRIPTION GENERATOR

by

BRUCE CHARLES COCEK, B.A.

August, 1974 TR-41

This paper constituted in part the author's thesis for the M.A. degree at The University of Texas at Austin, August 1974.

Technical Report 41

THE UNIVERSITY OF TEXAS AT AUSTIN

DEPARTMENT OF COMPUTER SCIENCES

A C K N O W L E D G M E N T S

I would like to extend my appreciation to Dr. Terrence W. Pratt for his help, guidance, and understanding during the supervision of this report. Also, I would like to extend my appreciation to Dr. John H. Howard, who served as reader.

I would like to thank my family and friends for their encouragement and patience during the preparation of this report.

B. C. C.

The University of Texas at Austin

July 15, 1974

TABLE OF CONTENTS

Chapter		Page
I.	INTRODUCTION	1
II.	CONSTRUCTION	14
III.	DERIVATION	39
IV.	EXAMPLES	54
V.	IMPLEMENTATION	74
VI.	CONCLUSION	82
APPENDIX	. . . . .	85

C H A P T E R I

INTRODUCTION

The goal of this study is to provide a means by which a case description for a given program may be derived. The program which has been written to handle this does so in a concise and efficient manner. The results of this study may then be used to achieve several goals. Among these goals are proofs of the correctness of programs, analysis and debugging of programs, equivalence proofs, and program synthesis. The first two of these applications are of particular interest to the practical user of computing machinery.

Background

Every program has the characteristic that it may be divided into two segments, the control and the kernel, which are able to describe fully the flow through the program and the output produced by that program. This idea was introduced by Pratt (4) in his paper on kernel equivalence. In this work, Pratt gives workable definitions for the kernel statement and the control statement. He defines

kernel statements as statements within a program which "participate directly in the computation of some output" and control statements as statements within a program which "participate directly in deciding the control path at branch points." He also shows the relationship between these two classes of statements in proving kernel equivalence of programs. The reader is directed to this work for a more detailed description of kernel equivalence.

The case description was proposed in another work by Pratt (5) in which he continues the study of the kernel and control segments of a program. Pratt's paper gives a detailed account of the case description and its derivation. He also provides an algorithm with which a person is able to determine case descriptions given a flowchart as input. The following paragraphs provide a brief explanation of program graphs and case descriptions.

Program Graphs

It is assumed in this study that every program may be represented by a flowchart, consisting of a directed graph with labelled nodes and arcs. This flowchart is a single entity, with all nodes being connected in some manner by arcs. Each flowchart is assumed to be correct, that

is, it is assumed that the flowchart meets the specifications for which the given program is intended. This flowchart then becomes the basic model used throughout this study. This model is a representation that has become widely used (see e.g., Luckham, Park, and Patterson (3), Kaplan (2)).

In our model, nodes may have zero, one, or two outgoing arcs. These nodes are labelled exit nodes, assignment or entry nodes, and branch nodes respectively. The entry node is unique in that it is the only node in the graph with no incoming arcs. Exit nodes have at least one incoming and no outgoing arcs. There may be more than one exit node in the graph. Assignment nodes and branch nodes have one or more incoming arcs but at least one of these incoming arcs must be from another node. Each and every node in the graph lies along some path from the entry node to an exit node. However, this does not mean that every path taken through the model will end in an exit node (e.g. paths which generate infinite loops will never reach an exit node). Branch nodes are defined as having two outgoing arcs labelled "T" (true) and "F" (false). These arcs are referred to as the true branch and the false branch, respectively, when used in the description of the implementation. In the model, assignment nodes are labelled with

assignment statements, brach nodes are labelled with branch statements, the entry node is labelled "ENTRY," and the exit nodes are labelled "EXIT." From the model constructed in such a manner, we are able to derive the control and kernel segments necessary to construct the descriptors in the case description. Figure 1.1 depicts a basic program graph. In this example, the types of nodes are labelled as assignment, branch, entry, or exit for clarity.

Case Descriptions

A case description is comprised of a set of descriptors, each containing a control expression and a kernel expression. The control expression of the descriptor specifies a set of conditions which the input to the program must satisfy in order to achieve a desired output. The control expression is derived from a path through the program and corresponds to the control statements encountered in this path through the program. There is no ordering to the terms in the expression since the input must satisfy all terms within the expression in order to achieve the desired output from the program. An example of a control expression is shown in Figure 1.2.

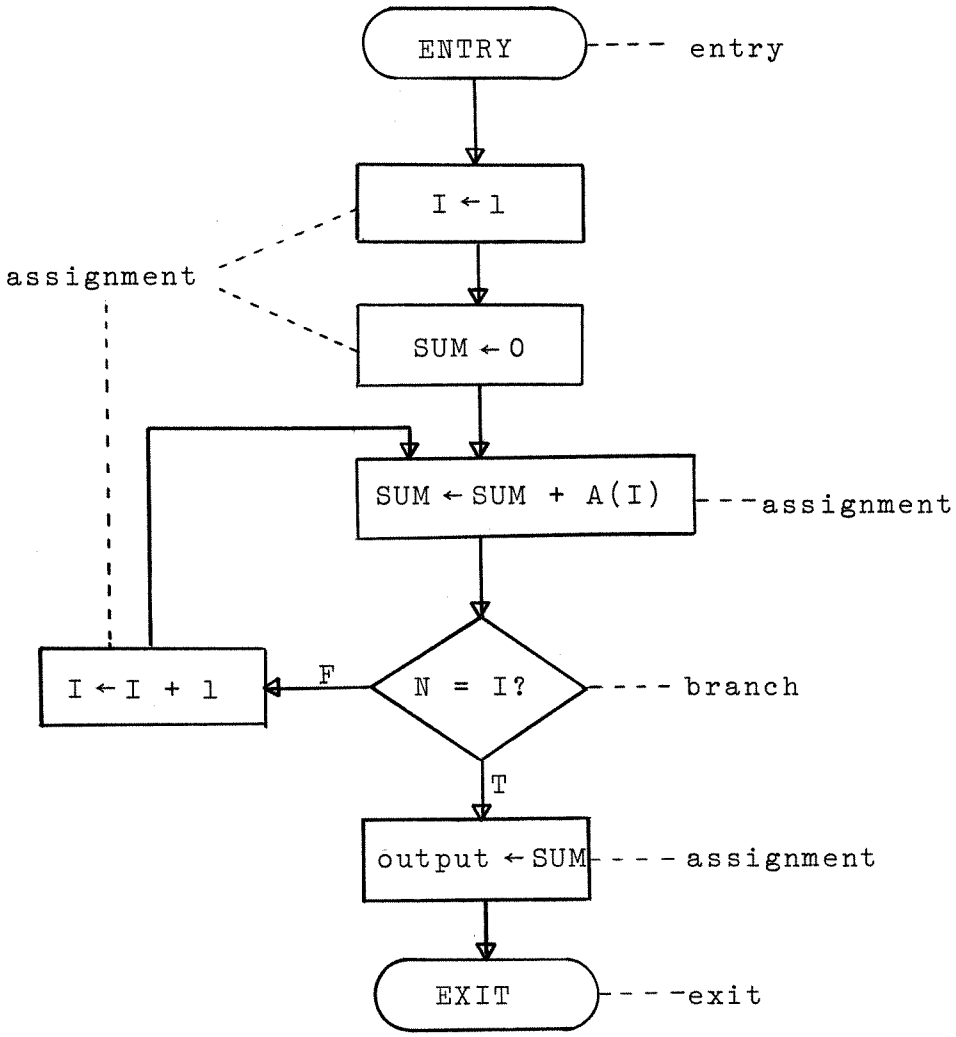

ENTRY
entry
I ← 1
assignment
SUM ← 0
SUM ← SUM + A(I)
assignment
I ← I + 1
F
N = I?
branch
T
output ← SUM
assignment
EXIT
exit

FIGURE 1.1

$$C = (\sim(N = 1), \sim(N = 1 + 1),\ N = (1+1) + 1)$$

Figure 1.2

The second half of the descriptor is the kernel expression. The kernel expression specifies the output of the program when program inputs satisfy the control expression. A kernel expression in final form is depicted in Figure 1.3

$$K = (\text{output} \leftarrow ((0 + A(1)) + A(1 + 1)) + A((1 + 1) + 1))$$

Figure 1.3

These individual descriptors are then grouped in order to form the case description for the program. The case description takes the following form:

if (control expression$_1$) then (kernel expression$_1$) else
if (control expression$_2$) then (kernel expression$_2$) else
.
.
.
if (control expression$_n$) then (kernel expression$_n$)

with three major exceptions: (1) the descriptors are unordered, (2) for any given set of input values, at most, one

control expression is satisfied, and (3) there may be an infinite set of descriptors.

The case description of Figure 1.4 is for the program of Figure 1.1 which computes the sum of the elements of an array. This case description does not contain every possible descriptor for this particular program since the descriptors are infinite in number. It does provide, however, the general form in which the case description appears.

```
C = (N = 1)
K = (output←0 + A(1))
C = (~(N = 1), N= 1 + 1)
K = (output←(0 + A(1) + A(1 +1))
C = (~(N = 1), ~(N = 1+ 1), N = (1 + 1) + 1)
K = (output←((0 + A(1)) + A(1 + 1)) + A((1 +1) +1))
                        .
                        .
                        .
```

Figure 1.4

In his paper concerning case descriptions, Pratt provides a means for deriving a descriptor. Each descriptor is derived from some path through the program graph for a

given program. For each predicate node encountered in the sequence of statements from the entry point to the exit point, the predicate statement or its negation, depending on whether the true or false branch is followed, is included. Each statement in this sequence is either an assignment, a predicate, or a negation of a predicate. This sequence is known as the unreduced descriptor. The algorithm provided by Pratt to derive a descriptor, composed of a control expression C and a kernel expression K, from this unreduced descriptor is as follows:

1. Choose the first statement in the unreduced descriptor.
2. If an assignment statement, e.g. "V←F(...)", then:
   (a) In each following assignment and predicate in the unreduced descriptor in which the variable V occurs as an argument, up to and including the next assignment to the same variable V, if any, replace each occurrence of V by the expression on the right of the ← in the original assignment (enclosing the expression in parentheses to avoid ambiguity if necessary). Of course, the final occurrence of V if any, on the left of the arrow in the last assignment statement is excluded.

(b) Delete the original assignment from the unreduced descriptor. If the variable V is in the output set of the program and no later assignment to V occurred in the descriptor, then place the deleted assignment in the kernel expression, K.

3. If the leftmost statement is a predicate or the negation of a predicate, place it into the control expression, C, and delete it from the unreduced descriptor.

4. Repeat steps 1-3 for each statement in the unreduced descriptor.

The reduced descriptor is the pair (C,K) of expressions resulting from the reduction process.

Derivation Tree

The implementation of a case description generator presents a need for an alternative representation of a program. This representation must enable us to derive information concerning every possible path through the program. From this representation we must also be able to extract a suitable idea of the results which may be obtained by following the flow of the program from the entry point

to some exit point within the program. From the information concerning the paths through the program derived from this representation, we are able to generate the control expression of our descriptors. Likewise, from the information concerning the results obtained following these paths, we are able to construct the kernel expression of our descriptor. Thus, through repetitious use of this representation, our case description unfolds.

For this representation of programs, we define a new type of tree structure. This structure is derived from a basic binary tree, i.e., a tree structure in which each node has exactly two descendants. The two arcs leaving each node will correspond, respectively, to the true and false branch of a control statement. The definition of a derivation tree is as follows:

> DEFINITION: A derivation tree is a finite set of nodes connected by directed arcs, which satisfy the following conditions (if an arc is directed from node 1 to node 2, we say that the arc is an outgoing arc from node 1 and an incoming arc into node 2):
>
> 1. There is exactly one node into which there are no incoming arcs. This node is called the root. The root has only one outgoing arc.
>
> 2. For each node in the tree there exists a sequence of directed arcs from the root to the node. Thus the tree is connected.

3. The number of incoming arcs to any node except the root is unlimited. The node in question is a direct descendant of the nodes from which these incoming arcs originate.

4. Each node has, at most, two outgoing arcs.

This definition differs from that of a binary tree in that looping is allowed within this tree. The reason for this is that, without this provision, the derivation tree could reach outlandish proportions at a rapid rate and would soon become unfeasible as a means of representation of a program. The allowance of looping within the tree permits the inclusion of more information concerning the structure of the program without the exclusion of important facts. An example of a binary tree is shown in Figure 1.5 and a derivation tree is shown in Figure 1.6. Other definitions which are used in this study follow.

DEFINITION: The set of all nodes n, such that there is an outgoing arc from a given node m into a node n, is called the set of direct descendants of m.

DEFINITION: A node n is called a descendant of node m if there is a sequence of nodes n_1, n_2, n_3, . . ., n_k such that $n_k = n$, $n_1 = m$, and for each i, n_{i+1} is a direct descendant of n_i. We shall, by convention, say a node is a descendant of itself.

DEFINITION: Some of the nodes in any tree have no descendants. These are known as the exit nodes

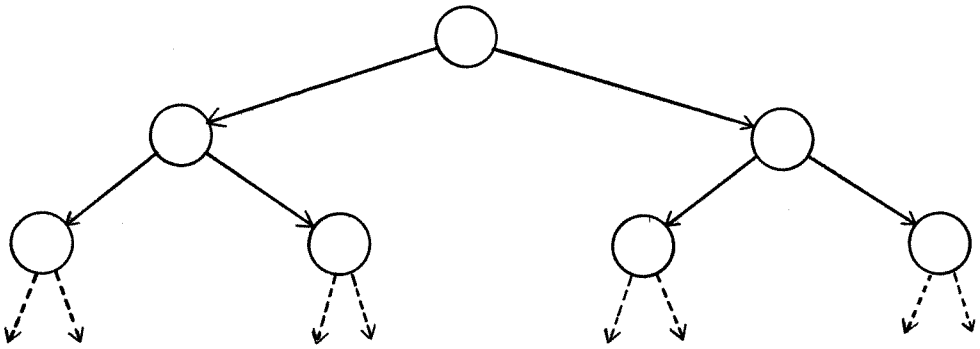


FIGURE 1.5

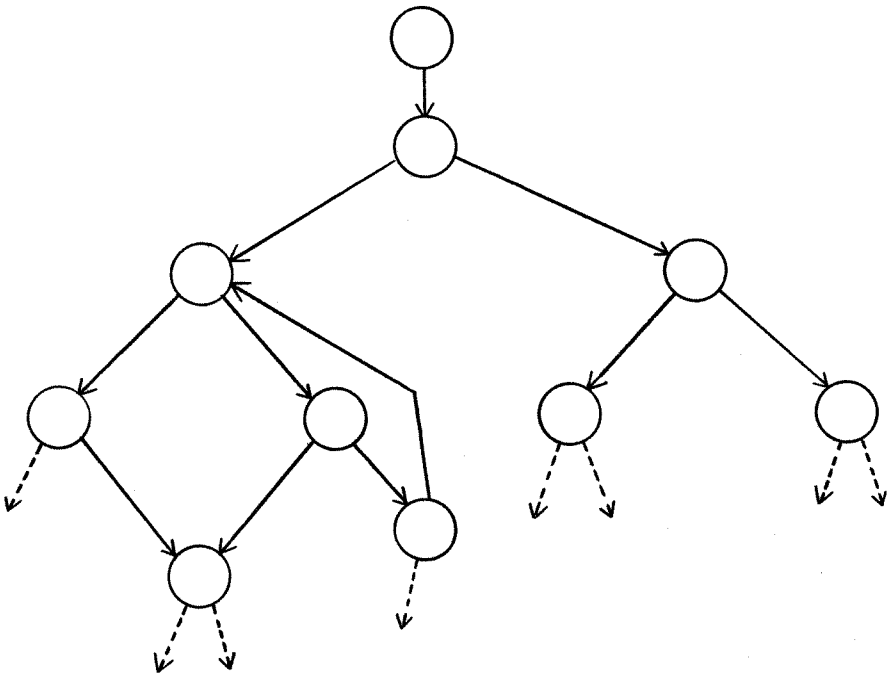


FIGURE 1.6

within our tree and provide the end points of paths through the tree.

DEFINITION: A subtree within a given tree is a particular node of the tree together with all its descendants, the arcs connecting them, and their labels.

Summary

Chapter II of this study shows how a derivation tree is constructed for a program. It describes the algorithms needed for this construction and how the tree is developed from the input graph structure.

Chapter III describes the derivation of a case description from the derivation tree built in Chapter II. It shows how different types of paths are handled and how the control and kernel expressions for each descriptor are built.

Chapter IV gives an example of the derivation of a case description in detail. This chapter uses the Wensley Division Algorithm as an example and performs the necessary action upon its graph representation in order to obtain its case description.

Chapter V describes the program which has been written to implement the case description generator.

Chapter VI is the concluding chapter of the study.

C H A P T E R I I

CONSTRUCTION

We are now ready to begin construction of the tree used in our case description generator. This chapter explains that construction, beginning with the basic structure consisting of branch nodes and directed arcs and then adding new features to the structure making the derivation of the case description possible.

Input Graph

The input to our case description generator consists of programs in graph form. To get programs into this form, a module is required which is able to perform the translation of programs from their basic form, i.e., a set of statements, into acceptable input for the generator, i.e., a graph which is, in fact, a flowchart for that program. The module to perform this task has been proposed by Pratt (6). The graph produced by this module has the characteristics described in Chapter I. From here on, we assume our input to the generator is the graph, G, and the program it represents is known as P.

The input, G, being in flowchart form, consists of two types of nodes, branch nodes and assignment nodes. Each of these two types of nodes has been tagged prior to input to the generator. These tags denote whether the node represents a control statement (C), a kernel statement (K), or both a control and kernel statement (CK), or neither a control or a kernel statement ( ). An example of an input graph is shown in Figure 2.1.

In the first part of the construction, we are interested chiefly in the branch statements. Hence, the first step in the generation of the case description provides some means other than the number of outgoing arcs to distinguish between these two types of statements and also a means to distinguish entry and exit points in the program.

Initialization

The graph, G, is set up in a central memory workspace. This graph takes up only a small portion of the workspace. The remaining portion of the workspace is used to build the tree structure utilized by the case description generator. The location of each node within G is known to the generator when the graph is set up in the workspace.

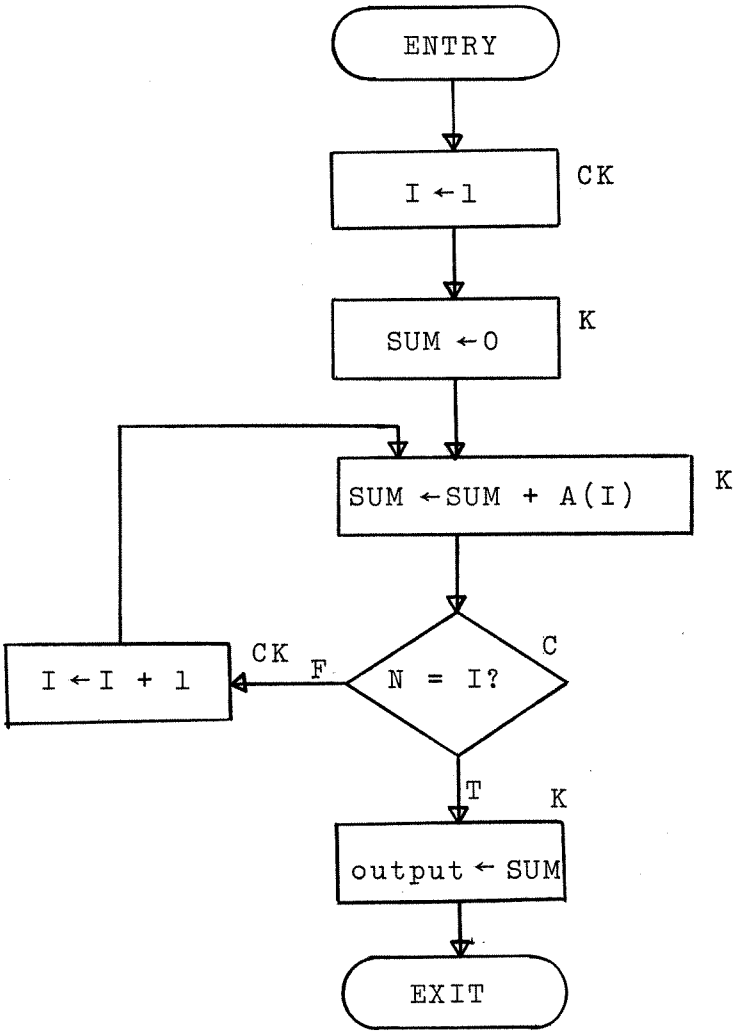


FIGURE 2.1

The generator makes a preliminary pass of G and labels each node within the graph according to type. Entry nodes are labelled with 'A' denoting a starting point, exit nodes are labelled with 'Z' denoting a terminal point, branch nodes are labelled with 'B' denoting branch statements, and assignment nodes are labelled with 'F' denoting assignment statements.

Also during this preliminary pass of the graph, the generator sets up an Assignment Table containing labels which are to be assigned to each node in G to enable easier referencing. This table is an array with each element composed of three entries. The first is the statement in the program which the node represents, the second is the type which is assigned in this first pass, and the third is the tag with which the node was input to the generator. The label by which the node may now be referenced is the subscript used to access its corresponding entry in the table. The construction of this table is accomplished according to the following algorithm:

1. Y ← 1
2. Find the entry node in the graph. Move "ENTRY" to the statement portion of the table indexed by

Y; move a blank to the tag portion of the table indexed by Y; replace the statement which labels the node in the input graph with the value of Y.

3. Find a node in G which has not been referenced previously. If no such node exists, the algorithm has completed its processing.
4. Y ← Y + 1
5. Move the statement in the node to the statement portion of the table indexed by Y; move the type (B, F, or Z) to the type portion of the table indexed by Y; move the tag for this node (C, K, CK, or a blank) to the tag portion of the table indexed by Y; replace the statement which labels the node in the input graph with the value of Y.
6. Go to 3.

When the algorithm has been completed, the preliminary pass is complete. As can easily be seen, the assignment of a position in the table is completely arbitrary since the assignment is used strictly for reference purposes and has no direct effect upon the outcome of the generation. A sample Assignment Table is provided in Table 2.1.

TABLE 2.1

Subscript	Statement	Type	Tag
1	ENTRY	A	
2	I ← 1	F	CK
3	SUM ← 0	F	K
4	SUM ← SUM + A(I)	F	K
5	N = I?	B	C
6	I ← I + 1	F	CK
7	output ← SUM	F	K
8	EXIT	Z	

This first pass of the graph, G, has built a table which proves to be extremely useful in our construction. It enables us, first of all, to use numeric labels on the nodes of our tree rather than awkward and sometimes bulky statements from which G is derived. For example, a statement such as IF(X.EQ.Y.OR.X.EQ.Z)... would be assigned to an arbitrary position x in the table. Instead of using this bulky expression to label the resulting node within our tree structure, we may now use the number x. Whenever the statement associated with this particular node is needed, it may be obtained by using x as a subscript to

reference the Assignment Table. It also enables us to shorten the control and kernel expressions which are constructed between branch statements (see below). Finally, it enables us to condense the amount of workspace needed for our tree structure.

Creation of the Root of the Derivation Tree

We now execute an initialization routine. This routine first sets up the workspace in which the tree is to be constructed and then creates the first node in the tree. This node becomes the root of the derivation tree. It is the only node in the tree which does not correspond directly to some branch statement in the program. It corresponds, instead, to the entry point of the program and it is not used in the derivation we make using the tree. Its primary use is to set up a point for inserting input specifications to the program.

The generator is now ready to make a second pass of the graph. In this pass, the tree from which the case descriptors are derived begins to be built. If the reader will recall, the tree structure which has previously been described consisted of nodes having two direct descendants

each. These nodes come directly from the branch nodes within G. Hence, this second pass is concerned only with the branch nodes of G, which are now distinguishable by the type assigned to them in the first pass.

This second pass of the graph, G, determines the first branch or exit statement, Q, encountered in G after the entry statement. If the statement represented by Q is a branch statement, a node is created to represent that statement and this statement or its negation becomes a part of all control expressions in this case description. If, however, Q is an exit point of the program, the graph represents a special case for the case description generator. In this case, the program contains no control statements and, hence, no control expression can be constructed.

If we have found some branch statement, we create a node to represent it in the tree and connect it to the root node with an arc from the root to this node. This node, I, is unique and distinguishable from other nodes in the tree in that it is the only node within the tree that is connected to the root node.

Determining Direct Descendants

We recall that each node in the definition of our tree has two outgoing arcs. Recalling also that each branch statement in the graph has two outgoing arcs, we are able to equate the nodes of our tree with the branch statements in G. The construction of our tree then becomes a simple matter of translating the branch statements in the graph into the nodes of our tree.

The tree is constructed by levels. Each level is completed before advancement to another level. In this method, all nodes which are in the state of creation exist on a level which is at a given distance from the root. When all nodes have been created at this level, the distance is incremented and all nodes at this new distance from the root are created. Using this method, the tree grows steadily and all paths through the program are represented in the tree.

Hence, each node is assigned a level number according to its distance from the root. This distance is calculated by a count of the number of arcs crossed when traversing the path from the root node to this node. The root is on level 0 since no arc is crossed in the path

from the root node. The node representing the initial branch statement is on level 1. The next nodes encountered when traversing the path dictated by the true value of the initial branch statement and when traversing the path dictated by the false value of the same statement are both assigned to level 2. The direct descendants of these two nodes are assigned to level 3, etc. This assignment of levels continues throughout construction. Once a level number has been assigned to a node, this level remains the assignment of the node throughout execution. It is easily seen that some nodes could be encountered more than once, hence, could receive more than one level assignment. However, we shall limit the assignment to one level number, the initial assignment, no matter how many times the node is encountered in the path through G. The reasons behind this way of thinking are developed below.

In the use of our method, we must ascertain whether all nodes belonging to the present level have been created before advancing to the next level. When creation of the node representing the initial branch statement is complete, all nodes in level 1 have been created since this node is the only node in level 1. For now, this level is completed and we advance to level 2. To do this, we

must determine all direct descendants of all nodes in level 1, hence, the direct descendants of the initial branch node. This node, or any node, has, at most, two direct descendants, one corresponding to each of the paths determined by the true or false value assigned to the branch statement. Each direct descendant must be one of two types; it must be an exit node, in which case the path is complete upon creation of that node, or it must be another branch node which continues the tree.

We now determine the direct descendant, Q, of the initial branch node. If Q is an exit point of the program, a special node is created to represent it. This node is labelled "EXIT" and shows that the path through the program has reached a termination point. From this path, using the tree, we are able to determine a complete descriptor. How this is done is shown later.

If Q is another branch statement, the tree continues to grow. This statement also has two direct descendants which generate other nodes. We create a node to represent Q in the tree and label it with the node label assigned in the first pass.

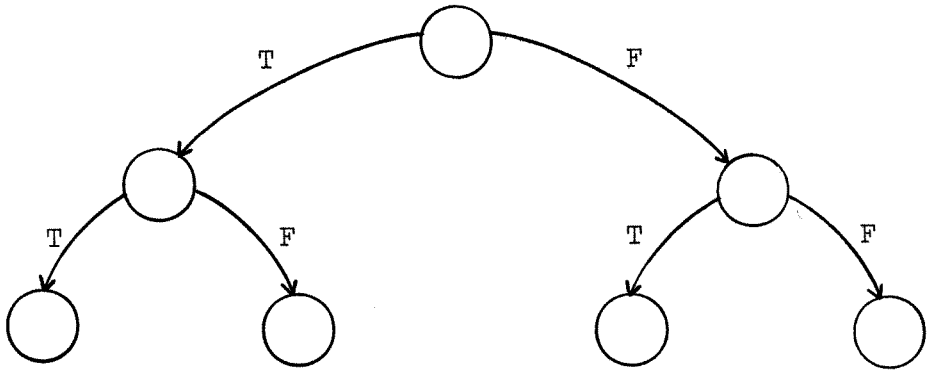
Connection of Direct Descendants

Now we have possibly two more nodes with which to proceed in the construction of our tree. We create an arc in the tree as an outgoing arc from the parent node and as incoming arc to the descendant node. The arc from the parent to the descendant is the primary connecting arc. These arcs are of great importance since they are used as the points of attachment for the tree control and tree kernel lists which are constructed between each node in the tree. An explanation of these lists is given later in the text.

There must be, for the structure to have meaning, some way to distinguish between the two arcs generated for each node. For this reason, as the arcs are created, they are given labels appropriate to the branches which they represent. True branches are labelled as 'T' and false branches are labelled as 'F'. A sample tree showing the connecting arcs is shown in Figure 2.2.

Completion of the Tree

With completion of the arcs between level 1 and level 2 and creation of the nodes of level 2, we have



——— Primary arc.

FIGURE 2.2

finished processing for level 1 of our tree structure. We are ready for completion of level 2. Level 2 is completed much in the same manner as level 1. There are two nodes in level 2 which must be processed in order to create level 3. The order in which these two nodes are processed has no effect upon the final construction of the tree. Hence, the processing may be as if each were the root of its own tree, following the procedure as was outlined above.

Processing continues on the tree until no more direct descendants exist. When this occurs, the basic tree is complete. However, this situation will never occur if there are one or more loops in the program, since the path which continues the loop will never encounter an exit statement. So, if allowed to do so, the creation of the tree will continue and the case description generator, itself, will be in a loop. Therefore, some means must be found so that programs which contain loops may be represented by complete trees. In essence, there needs to be a way to prune trees in order to eliminate unnecessary segments of the tree.

Processing of Loops

So far, in the creation of the tree, we have not permitted more than one incoming arc to any node. But, in the definition stated in Chapter I, we allow more than one incoming arc to a particular node. Because of this modification, we are able to prune our tree.

Ancestor Lists

In the construction of the tree, the only way to determine the ancestor nodes of any particular node beyond its parent would be to follow the path back up through the tree until the path runs out, i.e., until the root is encountered. However, should we need to know the ancestors of each node which we create, this could prove to be very time consuming and inefficient. The point to be made is that we need to know every ancestor of every node in order to determine when a looping situation occurs. Hence, we need a method which is both quick and efficient. For this purpose, we create ancestor lists for each node describing the path up to that node.

Each ancestor list consists of two lists, the first containing the branch nodes encountered in the path

up to the present node (branch list) and the second containing the arcs traversed in this path (arc list). Each list is ordered, with each branch node being concatenated onto the end of the branch list as it is encountered and with each arc being concatenated onto the arc list as it is traversed. Hence, the branch list has as its first member the initial branch node and as its last member the node being processed with each node representation listed sequentially between them. Also the arc list has as its first member the representation for the arc connecting the root to the initial branch node and as its last member the representation for the arc connecting the present node with its parent with all arcs between this node and the root listed sequentially. The method for determining these lists and how they are attached follows.

The creation of the list begins at the initial branch node. Looking at the path of the program up to this node, we find that there are no branch statements preceding the initial branch statement which this node represents. Hence, the branch list contains only the label of this node. Since only one arc has been crossed in the path from the root, this arc is the only member of the arc list. These lists become the basis for all such lists produced in the tree.

The finding of a direct descendant determines the destination of the connecting arc. Depending upon whether or not a node representing the direct descendant exists in the tree, creation of a new node may or may not take place. There is, however, some point to which the path must advance. These nodes, likewise, must each have ancestor lists in the tree.

When we find the direct descendant of a node, we first create the ancestor list for this direct descendant. This is done in the following manner:

1. A duplicate of the ancestor list of the parent node is created.
2. The label of the node being processed is concatenated onto the end of the new branch list. The representation of the connecting arc is concatenated onto the end of the new arc list. These lists are combined to form a new ancestor list.
3. This list is attached to the node it describes. This list is now the ancestor list for that node.

These lists are used in determining looping situations, and, more importantly, in the construction of descriptors.

Arc Classification

Recall that after creation of the initial branch node, or all nodes at level 1, we proceed to create its direct descendants, or all nodes at level 2. Recall also that the arcs connecting the parent node to its direct descendant were labelled 'T' representing true and 'F' representing false. We now classify these arcs into two categories: 1) those which do not generate loops, or *non-looping arcs*, and 2) those which generate loops, or *looping arcs*. We also say that the descendant node into which a looping arc is incoming is said to be a *loop base*. This classification system and these terms are used extensively in the pruning technique.

After an ancestor list has been created and attached to the node it describes, we need to determine if a looping situation has occurred. A looping situation occurs when, in a path through a program, a branch statement is encountered more than once. To determine if a looping situation has occurred, we perform a sequential search of the ancestor list using the label of the node (the last label in its own ancestor list) as the search argument. If the search is successful, the arc is classified

as a nonlooping arc. Also, if the search is successful, the descendant is added to the Loop Base Table. This table is a list of all loop bases in the program and is used in derivation.

The generator now determines whether or not a node corresponding to the direct descendant has been created. If the direct descendant is not found among the nodes which have already been created in the tree, a new node is created as was explained earlier and the list is attached to this node. If, however, the direct descendant is found among the nodes of the tree, there is no need to create a new node. Hence, there is only one node to represent each branch statement in G no matter how many times an individual branch statement is encountered in the various paths through G. This method eliminates duplicate nodes in the derivation tree, therefore reducing the size of the structure.

Whether or not a new node is created in the tree, an arc is created between the parent node and direct descendant and classified as was explained above. The ancestor list is attached to the tree node as usual, whether the looping situation exists or not. Because there is only one node in the tree to represent each branch statement

in the flowchart, there may be multiple ancestor lists attached to a single tree node, each determined by a path from the root to this node. Each individual ancestor list is needed on the tree node in order to determine all possible looping situations which may occur at that node or any of its descendants. Now, as the generator creates the arc connecting the descendant with its parent, it performs a sequential search of each of the descendant's ancestor lists using the descendant as the search argument in order to determine whether or not the looping situation occurs and classify the connecting arc. When multiple ancestor lists exist, the generator must perform a search on each of these lists in order to determine the existing situation.

Now, we have allowed the direct descendant of a node to be at a lower level in the tree than the node itself. Processing on this direct descendant could possibly already be completed. Hence, provision must be made to process this path to completion.

If the generator has found the direct descendant node at a lower lever in the tree and the arc connecting the parent node with the direct descendant is a nonlooping arc, there exists a condition which must be handled in a

special way. If the direct descendant has yet to be processed, then the ancestor list is attached to the node along with any other ancestor lists which might be attached there. If, however, the direct descendant has been processed, the generator must propagate this possible path through all of the descendants of this node. These lists are passed on exactly as any other ancestor list would be passed on. This process is continued until the generator finds no other direct descendants which have not been processed or no other nonloop paths exist. During this process, the generator continues to check the direct descendant against the current ancestor list for matches. If one is found, and the arc to this node from its parent has previously been classified as a nonlooping arc, the generator changes the classification of the arc to the looping variety and adds the node to the Loop Base Table.

Algorithm for Determining Direct Descendants

In the text up to this point, there has been mention of determining a direct descendant in specific instances but no general algorithm has been proposed to achieve this. In this section, an algorithm to do this is

introduced. This algorithm, the Direct Descendant algorithm, is used by the generator to determine direct descendants. The parameters passed to this algorithm are the graph, G, the node for which the direct descendant is sought, J, and the value, V, either true or false, which is to be assigned to the branch statement represented by J. It returns the direct descendant, Q, and two lists, the control list, C, and the kernel list, K. These lists contain the control statements and kernel statements, respectively, which are encountered between the parent and its direct descendant. The Direct Descendant algorithm is as follows:

1. Initially, set C and K equal to the empty list. If V is true concatenate the label of the parent onto C; otherwise, concatenate the negation of the parent onto C. Use V to determine the next node in G.
2. Obtain the next node, X, in G. If X is a branch or exit statement, set Q equal to X and terminate the algorithm; otherwise, go to 3.
3. Access the Assignment Table using X as the index.
4. If the tag entry in the table is 'C', concatenate X onto C; otherwise,

if the tag entry in the table is 'K', concatenate X onto K; otherwise,
if the tag entry in the table is 'CK', concatenate X onto both C and K; otherwise,
no action is taken.

5. Go to 2.

Control and Kernel Lists

So far, in the construction of the tree, we have dealt almost entirely with branch statements. The segment of the tree which we have constructed and the means by which we have constructed it is used in the construction of the control and kernel expressions of each descriptor. This section describes the segment of the tree which is used for the derivation of these expressions.

In the previous section, we saw that when the generator sought the direct descendant of a particular node in the tree along a given path, it created 2 lists of statements encountered when traversing the path defined in the graph representation, a control list and a kernel list. Each statement which is encountered along the path through G causes the generator to concatenate its label representation onto the appropriate list. These lists are complete when the next branch or exit statement is encountered.

In order to determine the kernel expression associated with each control expression in a particular descriptor, we need to know the exact kernel statements in a given path through the program. The kernel lists which the generator has created provide these kernel statements in a segmented fashion, i.e., they describe which operations have been performed between branch statements in the path from which the particular descriptor is derived. Hence, these lists may be utilized in determining the kernel expression of a descriptor. In a similar manner, the control lists returned by the Direct Descendant algorithm show the control statements encountered and may be used to determine the control expression of the descriptor.

The most logical and, likewise, the most efficient location of attachment for these lists is to place them exactly where they would be in the flow of the program, i.e., the arcs which connect nodes in the tree. Hence, the generator attaches the list of control statements and the list of kernel statements upon the arc making it truly representative of the path between statements represented by nodes in the tree.

Although this segment of the tree has not been brought forth until now, its creation takes place at the

same time as the branch segment of the tree. As the generator seeks the direct descendant of a node, the lists which are built are saved for this segment of the tree. When, eventually, the generator does find a direct descendant, it creates a node to represent this statement and an arc to connect this node to its parent. At this time, these lists are attached to the primary arc connecting the two nodes. The initial branch node is treated exactly as other nodes in the tree. Its lists are attached to the arc connecting it with the root node created at the beginning of construction. These lists may have any number of elements and may be the null, or empty, list.

Construction is now complete. In this chapter, we have shown how a graph representation of a program is transformed into the case description derivation tree. From this tree, we are able to generate individual control expressions and their accompanying kernel expressions which are used to construct the case description for the program it represents.

C H A P T E R I I I

DERIVATION

The case description generator has, so far, produced the tree with which we derive the individual descriptors. This chapter explains in detail how a descriptor is generated from the case description derivation tree.

Description of Output

The output for the case description generator consists of a set of descriptors for possible paths through a program. Each descriptor consists of two lists, one corresponding to the control expression of a descriptor and the other corresponding to the kernel expression of a descriptor. These lists should not be confused with the control and kernel lists which are attached to the arcs connecting two nodes in the tree. The lists attached to the arcs in the tree shall be known as tree control and tree kernel lists in this chapter; the lists used to build the control and kernel expressions shall be known as the control and kernel lists, respectively.

Briefly, these control and kernel lists are constructed in the following manner. Each list starts as an empty list. In a given path through a program, certain control and kernel statements are encountered in the program and the same statements are encountered in the corresponding path through the tree. As these statements are encountered, their label representations are concatenated onto the appropriate list in the form corresponding to the path through the program. Kernel statements are concatenated onto the kernel list as they are encountered. Likewise, control statements are concatenated onto the control list as they are encountered. When an exit statement is encountered, the descriptor is complete.

Although it is possible to derive descriptors from a graph representation of a program, the tree allows us to find paths in the program which generate meaningful descriptors more easily.

Within a given program, there are usually an infinite number of paths from the entry point to some exit point in that program. If the program contains no loops, the number of paths is finite. If, however, the program contains one or more loops, the number of possible paths becomes infinite. Since each descriptor is derived from

an individual path through the program, the case description for a program with loops also becomes infinite. To try to derive a case descriptor for such a program would be impossible without some limitations on the size of the case description.

Loops as Interruptions in Program Flow

No matter what type of program we are processing, it contains only a finite number of paths which contain no loops at all. A path which includes a loop also includes a path which contains no loops at all in some form which does reach an exit point in the program. This path may be interrupted at some point by a loop, but when the interruption due to the loop is complete, the path resumed conforms to a nonloop path. Hence, any path through a program which has had all loops removed from it conforms to some nonloop path. Therefore, we think of loops as being interruptions which may be inserted into any nonloop path in order to obtain some path through the program.

Loop Descriptors

We recall that a node which has an incoming arc from a node below it in the tree is known as a loop base. These loop bases are used as points at which to insert the interruptions determined by a loop. We use these loop bases as both the start and end points of any loop insertion into a path through the program. The insertions are, in essence, small loop descriptors which may be inserted into another descriptor at the point at which the loop base is encountered in a nonloop path. These loop descriptors are descriptors for one cycle of the loop but may be repeated any number of times in order to produce the desired descriptor. Hence, in our derivation, we process all loop paths first. Later, when we are processing the set of nonloop paths, we provide points of insertion in each descriptor for these loop descriptors.

During construction of the tree, a Loop Base Table was built containing all loop base nodes in the tree. This table is made use of at this time. On the nodes represented in this table, we find one or more ancestor list, each containing lists of labels for branch nodes and for arcs. We may determine from these lists the total number of loops within the program.

Loop Processing

Processing begins with the first loop base in the Loop Base Table and continues until all loops have been processed. Each loop within a program has a defined path for one cycle of the loop. This path is determined by some ancestor list for that loop base which is associated with the loop being processed. From these lists, we are able to determine the arcs which are traversed in one cycle of the loop.

We begin processing of the ancestor lists by searching the arc list of each ancestor list. If the loop base is a node from which one of the arcs is outgoing, we know that this particular ancestor list determines a looping path which originates at the loop base. If the loop base is not a node from which an arc in the arc list is outgoing, this ancestor list does not determine a looping path and we continue processing with other ancestor lists for this loop base. When all ancestor lists for a given loop base have been processed, the generator has completed processing for that loop base.

When an ancestor list is found which determines a looping situation, the generator makes use of the arc list within the ancestor list to determine the descriptor

for the loop. A search is performed upon the arc list until a representation of an arc which is outgoing from the loop base is found. This arc is used to determine the control and kernel lists for the loop. Originally empty, the control and kernel lists have concatenated onto them the tree control and tree kernel lists, respectively, which are attached to the connecting arc. Each arc represented in the arc list is processed sequentially in the same manner until the arc list is exhausted. When this occurs, the descriptor corresponding to the particular ancestor list is complete and the generator processes other ancestor lists for the loop base.

The generator processes all ancestor lists for a given loop base in the same way. Processing continues in the same manner for all loop bases in the tree. When all loop bases have been processed, the generator has completed processing for loops and is ready to process direct, or nonlooping, paths through the program. However, an explanation of how loops are to be inserted is given first.

Method of Insertion of Loops

It is possible that the direct descendant determined by an arc may also be a loop base. If this is the case, we must somehow show that another loop may be inserted at this point. The means by which this is done is outlined in the following paragraph. This method is used throughout derivation to designate insertion points for a loop or loops into a path through the program.

As we are now processing, we generate a single descriptor for each loop path. This loop descriptor describes the path of one cycle through the loop. As was explained above, these descriptors may be inserted into any path containing the loop base and may be inserted any number of times. To signify the point at which a loop may be inserted into a descriptor, we concatenate a label of the form LOOP x onto the control list and kernel list after concatenation of the tree control and tree kernel lists from the incoming arc onto the control and kernel lists. The letter x signifies the subscript with which the loop base is accessed in the Assignment Table. This expression denotes that the loop originating at the particular loop base may be inserted into this descriptor any number of times to determine a new path.

Any node may be the loop base for more than one loop. Hence, any one of these loops may be inserted into a path at the point where a loop insertion is signified.

Nonlooping Paths

Processing of nonloop paths is done as the final step. Whereas, in loop processing, we sequentially followed the path dictated by the ancestor lists, in the method which has been chosen for use for nonloop paths, we first find the goal which we seek and then determine the means by which this goal was achieved. In our situation, we determine the exit points of the program and then find the shortest paths through the program which may be traversed in order to reach these exit points.

Seeking exit nodes in the tree, we begin a search of the Assignment Table looking for exit nodes (type = Z). Our search continues until at least one exit node is found. The first level which contains such nodes produces the shortest paths through the program. (Recall that length of path was defined by the number of branch statements which were encountered in the path. A path which encounters an exit node at a lower level than other paths

encounters fewer branch statements and, therefore, is of shorter length.) Each exit node is processed to obtain all possible paths not containing loops which terminate at the particular level which is being processed. When all exit nodes have been processed in such a manner, the case description generator has found all possible nonloop paths through the program. An example of the order of processing is shown by using Figure 3.2 and Table 3.1.

TABLE 3.1

Label	Type
1	A
2	I
3	B
4	B
5	B
6	Z
7	B
8	Z
9	B
10	Z
11	Z
12	Z

Figure 3.2 is the case description derivation tree constructed for some program and Table 3.1 is the Assignment Table for that program. Searching the table, the generator first finds the node represented by the label 6 to be an

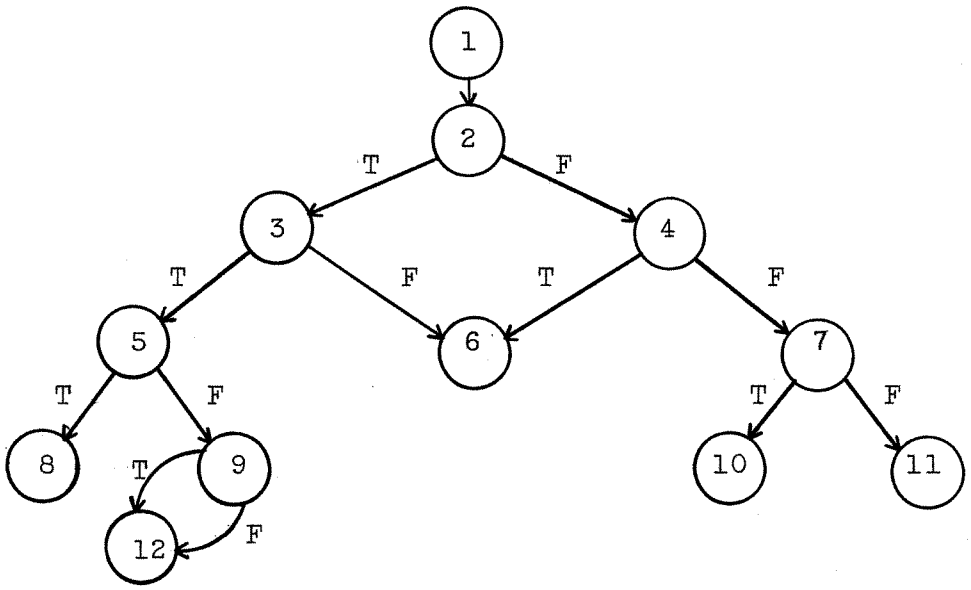


FIGURE 3.2

exit node. The generator processes the ancestor list of node 6 to find all paths from the root to this node. When all such paths have been processed, the generator returns to the table to find another exit node to be processed, this time returning with node 8. This node is processed in a similar manner as are nodes 10, 11, and 12. When the last descriptor has been built for node 12, the case description for that program is complete.

Processing of an Exit Node

For each possible path which ends at the exit node being processed, we create two lists. The first is a list of control statements and the second is a list of kernel statements. At creation for each path, these lists are empty.

Attached to each of the exit nodes in the tree are one or more ancestor lists which have been generated by particular paths through the program. Each of the lists describe a unique path through the tree from the root to this exit node, and also describe a unique path through the program for which the description is being built from the entry point to an exit point. Hence, these ancestor

lists are utilized to create the descriptors for the path from which they are generated.

The generator finds an ancestor list attached to the exit node which has not been processed. From this ancestor list, the generator extracts the arc list which is a list of all arcs which have been traversed in the path from the root to the exit node. The task for the generator now is to use this list to build the control and kernel expressions for the descriptor.

To accomplish this, the generator processes sequentially each member of the arc list. The generator accesses each arc in the list and uses the lists attached to these arcs to build the control and kernel expressions for the descriptor. The tree control list from the arc is concatenated onto the control expression and the tree kernel list from the arc is concatenated onto the kernel expression. Each arc in the arc list is processed in a similar manner. When all elements of the list have been processed, the descriptor is complete.

As was explained previously, loops may be considered as interruptions in the normal flow through the program. The loop descriptor for a loop may be inserted into a path when its loop base is encountered in the path.