Representing Control Constructs in Object-Flow Process Diagrams

Mor Peleg and Dov Dori

Faculty of Industrial Engineering and Management

Technion—Israel Institute of Technology

Haifa 32000, Israel

{mor, dori}@ie.technion.ac.il

Compositions of command statements, branching, iteration and recursion are primarily used in high-level programming languages to devise algorithms and obtain structured control flow. These structuring building blocks may also be instrumental for describing real-life systems in analysis and design methodologies at any complexity level. While composition and branching are incorporated into most prevalent analysis and design methodologies, they lack the capability of explicitly expressing iteration and recursion. We propose a method for incorporating case statements, iteration and recursion into Object-Process Diagrams, which are the visual formalism used in the Object-Process Methodology. A detailed case study dealing with testing samples of raw metal powders used in an industrial process of manufacturing inserts by sintering technology demonstrates the mechanism of this visual formalism.

1. INTRODUCTION

Compositions of command statements, conditional statements, iteration and recursion are primarily recognized as methods used by algorithms and programming languages in order to obtain structured control flow [1, 2]. These structuring methods may as well be effectively applied by analysis and design methodologies to describe real-life systems. Many graphical structured methods are widely used in order to analyze and design systems. Among the most prevalent are Coad and Yourdon's Object Oriented Analysis [3], Object Modeling Technique of Rumbaugh et al. [4], Jacobson's Object Oriented Software Analysis [5], Shlaer and Mellor's

Object Life Cycles [6], Booch's Object Oriented Development [7], Harel's Statecharts [8], Embley, Kurtz and Woodfield's Object Oriented System Development [9], Booch, Rumbaugh and Jacobson's Unified Modeling Language [10], and Firesmith, Henderson-Sellers and Graham's Open Modeling Language [11]. These analysis methods have a means of recursively expressing composition of lower level entities. These entities, which are expressed within statements, can be objects, processes, or object states. Conditional statements (if-then, if-then-else and case statements) can be represented by these methods by expressing more than one transition leaving a state in Statecharts or the Life-Cycles model. Even loops can be expressed in State Diagrams, although there are no built-in mechanisms for explicitly expressing bounded for-loops. However, none of the above mentioned analysis methods have adequate means for expressing recursive processes. Jackson System Development (JSD) [12] is a structured design methodology, which is based on functional decomposition. The system behavior is expressed through graphic diagrams accompanied by pseudocode, which, although can include iteration and recursion, can make JSD complex and difficult to fully comprehend than data flow and object oriented approaches.

The Object-Process Methodology (OPM) [13] is a visual formalism which incorporates the static-structural and dynamic-procedural aspects of a system into a single unified model. OPM achieves this by treating both objects and processes as complementary entities that together describe systems' structure and behavior in a single unifying model expressed graphically by a set of Object-Process Diagrams. OPM handles complex systems by using recursive seamless scaling [14]. It is suitable both for system analysis and system design, and enables smooth transition between theses phases [15]. Object-Process Analysis has been successfully applied in a variety of domains, including studyware design, database design, automating engineering drawing understanding, and computer integrated manufacturing [16].

In OPM, objects are viewed as persistent entities interacting with each other through processes—transient entities that affect objects by changing their state. Object-Process Diagrams (OPDs) enable us to describe objects and processes and how they interact with each other. Things (objects or processes) can be simple or compound. Compound objects [processes] are objects [processes] which are either a generalization of other objects [processes], or an aggregation of other objects [processes], or are characterized by other objects and/or processes. Compound objects and processes therefore enable statement composition, thus serving as control flow constructs. Objects may serve as enablers—instruments or intelligent agents—involved in a process without changing their state, or they may themselves be affected (or generated or consumed) by processes. The time-line in an OPD is directed from the top of the diagram to its bottom, so processes in an OPD are performed

in a top to bottom order, with concurrent processes located at the same height. A process can take place only if all the objects participating in it either as enablers or affected objects exist, and each is in its required state. A process can also be invoked by another process. The default logical relationship between links connecting different (participating as well as resulting) objects to a single process is the AND relationship. XOR and OR relationships can also be graphically expressed when required. Branching statements are expressed in OPDs by procedural (effect or enabling) links emanating from different states of an object. Each link connects the state to a process, as shown in Figure 1.

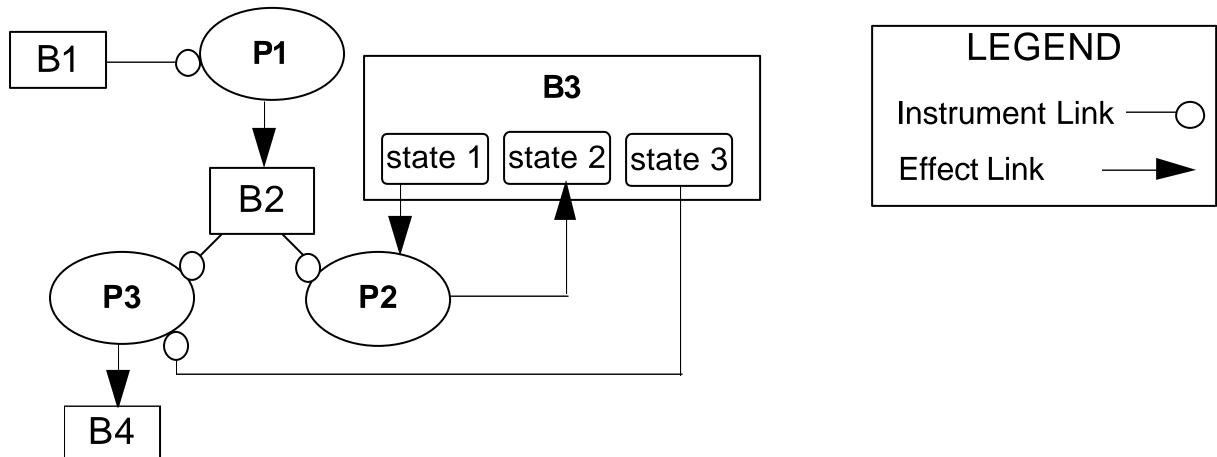

Figure 1: an OPD representing a conditional statement. The state of B3 determines whether P2, P3, or no process occurs.

The semantics expressed in Figure 1 is as follows. After process P1 is performed and object B2 is generated, depending on the state of object B3, either process P2 occurs, process P3 occurs, or no process occurs. If B3 is in state S1, B3 is affected by process P2 such that its state after P2 occurs is S2. If B3 is in state S3, B3 is an instrument for process P3, which results in the object B4.

This paper focuses on extending the expressive power of Object-Process Diagrams by enabling them to describe two prevalent control flow constructs: iterations and recursion. The initial motivation for the design of this extension was the observation that these control structures are not merely ideas that apply to the world of algorithms and programming. Rather, they show up in detailed analysis of actual industrial scenarios, such as the one used as a case study in this paper. We have encountered these control structures while analyzing the details of determining the total quality index of mixed metal powder batches used in the hard metal cutting tools industry, before the powder batch is pressed and sintered.

The paper is organized as follows. Section 2 describes the details of the Metal Powder Quality Assurance case study, in which both iteration and recursion play major roles. Section 3 handles iteration and shows how it

is incorporated into Object-Process Diagrams. Section 4 is similarly devoted to recursion. To illustrate graphic representation of recursion we use a simple example of a recursive post ordering algorithm. In Section 5 an integrated graphic representation of recursion and iteration is applied to the case study. A discussion on the merits and shortcomings of representing these control flow constructs graphically vs. specifying them through the use of pseudo code concludes the paper.

2. THE METAL POWDER QUALITY ASSURANCE CASE STUDY

Our Study has been motivated by a practical real-life problem we encountered while applying the Object-Process Methodology to a Business Process Re-engineering project of the technological knowledge base of a large metal cutting tool manufacturer. The problem is to compute a quality index for a batch of metal powder mixture before it is pressed and sintered. Figure 2(a) is an OPD showing the metal cutting tool manufacturing process. Metal powder is converted to "ready to press" powder mixture through the process *Obtain "Ready to Press" Powder Mixture*, which consists of two subprocesses: *Mixing*, and *Obtain Q[M]*, which determines the Quality Index *Q[M]* of the Mixed Metal Powder, *M*. Next, the Mixed Metal Powder is converted into a Raw Inserts through the *Sintering, Pressing, and Quality Checking* process, and finaly, the final Inserts are generated by the *Surface Treatment, Quality Checking, and Packaging* process. In the *Quality Checking* process, mixed metal powder samples are tested for different quality properties. The powder checking processes involve a hierarchy of tests which are executed recursively. This test hierarchy is illustrated in Figure 2(b).

For a given Ready-to-Press powder mixture M, there is a number m_p of different Test Groups that are performed on the powder mixture. Each Test Group is designed to determine a certain aspect of **quality properties** of the powder mixture. These aspects include, among others, magnetic, hardness, and density quality properties. Each Test Group includes a number of different kinds of Test Series that determine the values of a specific set of **quality attributes** of the powder mixture, which, together, determine the quality properties of a given Test Group. For each Test Series, a constant, predetermined number of samples is drawn. Each sample is tested, and the **quality results** of these Test Samples are used to determine the value of each Test Series quality attribute value. The resulting

quality attribute values from the different kinds of Test Series in a given Test Group are used to calculate the quality property value for each Test Group, and these quality property values, in turn, are used to calculate the final **quality index** of the entire batch of powder mixture.

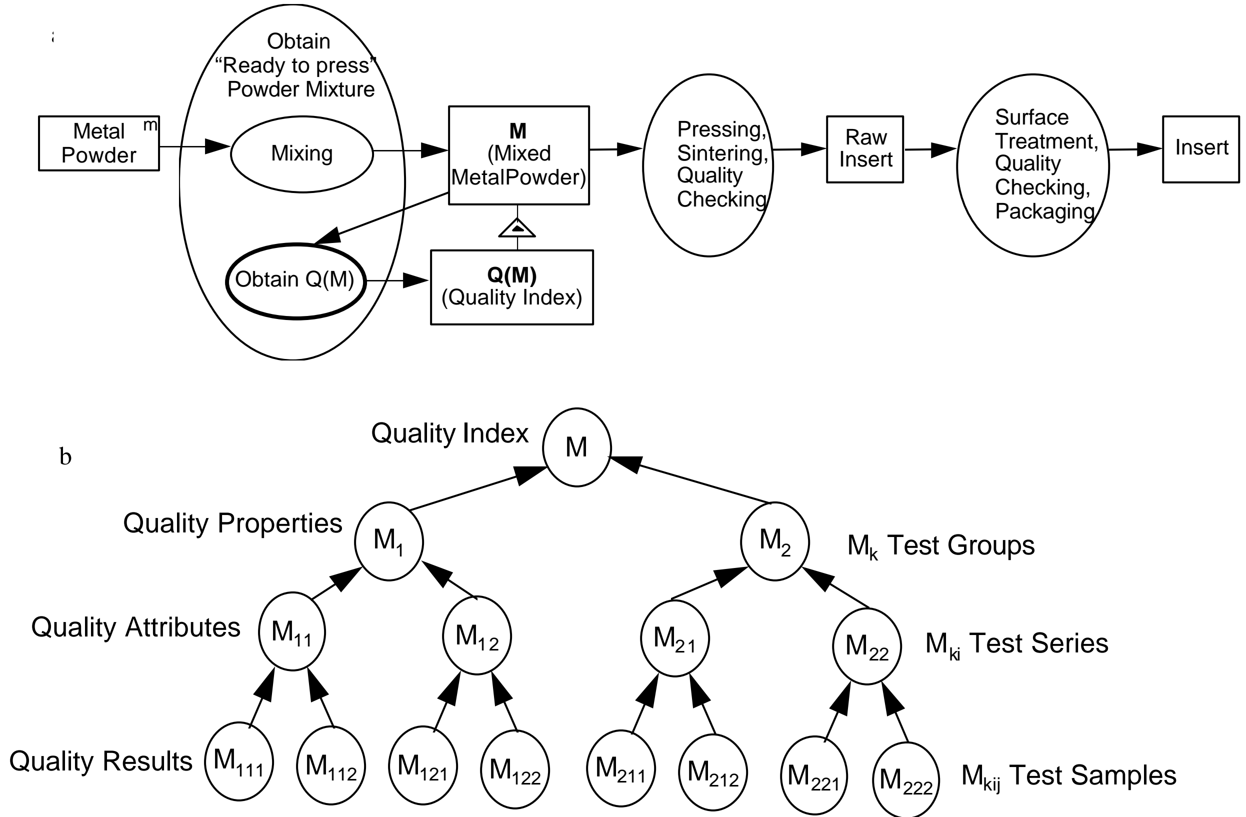

Figure 2: (a) an OPD showing the metal cutting tool manufacturing process. (b) The hierarchy of the metal powder quality assurance tests. "M" is the Ready-to-Press metal powder mixture. There are m_p Test Groups, each indexed by M_k, where k = 1,2,... m_p (Here m_p = 2). Each Test Group is divided into m_k Test Series, each indexed by M_{ki}, i= 1,2,... m_k . Each Test Series consists of m_i Test Samples, which are indexed by M_{kij}, j = 1,2,... m_i.

The processes of calculating the quality index of the metal powder mixture from the quality properties values of all the Test Groups, calculating the quality property value of each Test Group from the quality attribute values of its Test Series, and calculating the quality attribute value of each Test Series from the quality results of its Test Samples, are all executed iteratively at different levels in a bottom-up fashion.

The entire process of carrying out the quality tests is recursive in the sense that the first quality results are obtained from examining the actual Test Samples, which are the leaves of the quality

assurance test tree. These quality results are then combined in a specified calculation, to give a single result—the quality attribute value for each type of Test Series, and only then are the quality attribute values of the Test Series combined to yield the quality property value of each Test Group. Finally, the Quality Index of the powder mixture is determined using yet another specified calculation, using all the quality property values obtained from the various Test Groups.

3. ITERATION AND GRAPHIC REPRESENTATION OF LOOPS

Iteration is defined as repeating a process a number of times, until a set of stated conditions is met. Iteration is graphically denoted in flow-charts using loops, and the term "looping" has become synonymous with iteration. Loops can be divided into “for-loops” and “while-loops”. Both types have a loop body and a statement, which specifies the set of conditions required to terminate the loop. For-loops have a local counter, which is incremented each time the process executed in the loop body is repeated. The counter is compared to a constant, and when the relation between the counter and the constant satisfies the stopping condition of the loop, the execution of the loop body is terminated. The local counter may also be used by the loop body. While-loops are more general, as their stopping condition compares a variable, which is changed from within the loop body, to another variable or to a constant.

Figure 3 is a generic for-loop iteration Object-Process Diagram (OPD) [11]. The Iteration Entry and Exit Points are the initial and terminal objects used to connect the for-loop to the program's flow of control. The local counter of the loop is *j*, which is initialized to 1 by the process "*Initialize j*". *j* is compared to the stopping condition of the loop. The comparison results in exactly one out of the two possible states of *j*: *j* = stopping condition, or *j* <> stopping condition. If *j* is not equal to the stopping condition, the loop body is executed. Next, *j* is incremented and compared again to the number specified in the stopping condition. The state in which *j* is equal to the stopping condition number is the instrument for the Termination process, which results in the Exit Connection Object.

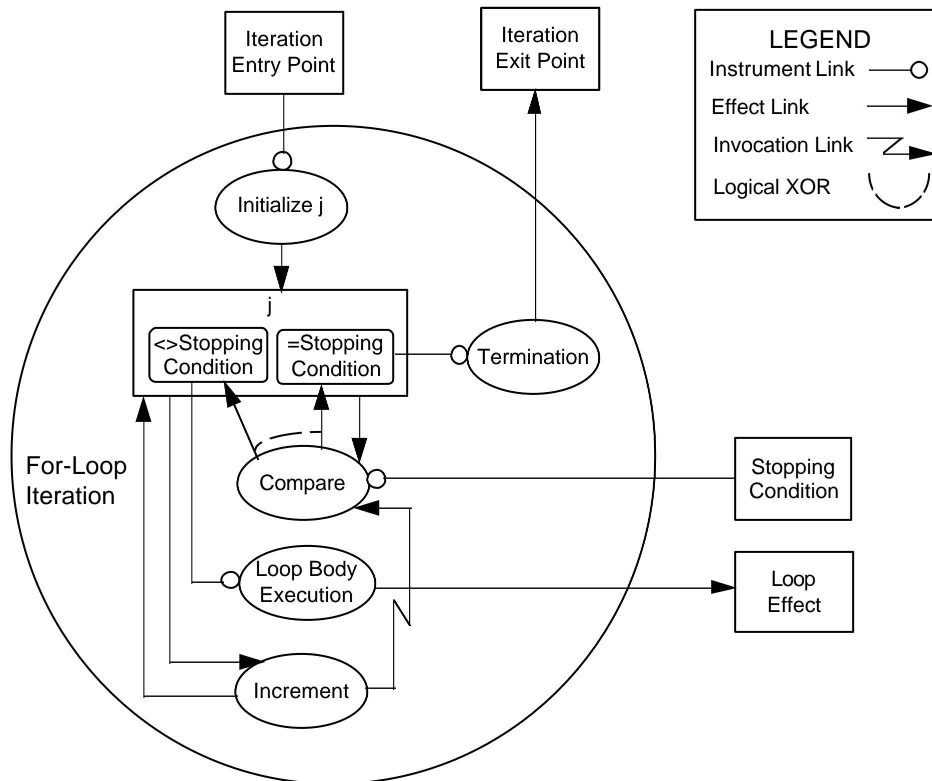

Figure 3: Generic for-loop iteration OPD. *j* is the local counter of the generic loop. The loop body is iterated until *j* equals to the stopping condition of the loop.

Since While-loops do not have a local counter, the above example would be implemented in almost the same way, apart from the fact that *j* would be a variable defined the object "*j*" and initialized outside the loop, and instead of the incrementation process, j would be affected by the *"Loop Body Execution"* process.

As noted, the need for representing loops graphically in the Object-Process Methodology has been raised while analyzing the process of powder testing which is a part of the quality assurance measures within the industrial process of manufacturing inserts by sintering technology. Figure 4 is an OPD depicting the process of obtaining the quality result $Q(M_{kij})$ of all m_i Test Samples j (j = 1,2,...m_i) of a powder mixture, M, undergoing a Test Series i in a Test Group k. The internal counter is represented

by j. The constant compared to j in the stopping condition is m_i, which is specific to each test type M_{ki}.

The first subprocess in the process *Obtain* $Q(M_{kij})$ is to initialize the loop counter j. Next, j is compared to the stopping condition m_i, which is an attribute of M_{ki} specifying the number of tests that should be performed on M_{ki}. The control enters exactly one of the two states $j \leq m_i$ or $j > m_i$. This is demonstrated by the XOR symbol between the two effect links. The branching of control flow is as follows. The state $j > m_i$ is the instrument for the termination process. The state $j \leq m_i$ is the instrument for the process *Calculate* $Q(M_{kij})$. The quality result of M_{ki}'s j^{th} sample is calculated by the loop body process *Calculate* $Q(M_{kij})$, and the result is output and assigned as the value of $Q(M_{kij})$. Next, j is incremented and compared again to m_i. This process is repeated until finally the control reaches state $j > m_i$, which is the instrument for the Termination process that leads to the Iteration Exit Point.

we will use the term "calling process" to refer to an instance of the recursive process that calls (or activates) another instance of the recursive process—the called process. A recursive process has a stopping condition, which, when met, causes the process to return (transfer the control) to the last call of the calling process. This implies that a recursive process has a "built in" memory mechanism. Indeed, programming languages, such as C, which support recursion, implement it usually by means of a stack, which "remembers" the last call as the top of the stack. Before describing the Object-Process Diagram of the recursive process of the metal powder mixtures quality tests, we describe in detail a simpler recursion, which does not involve loops.

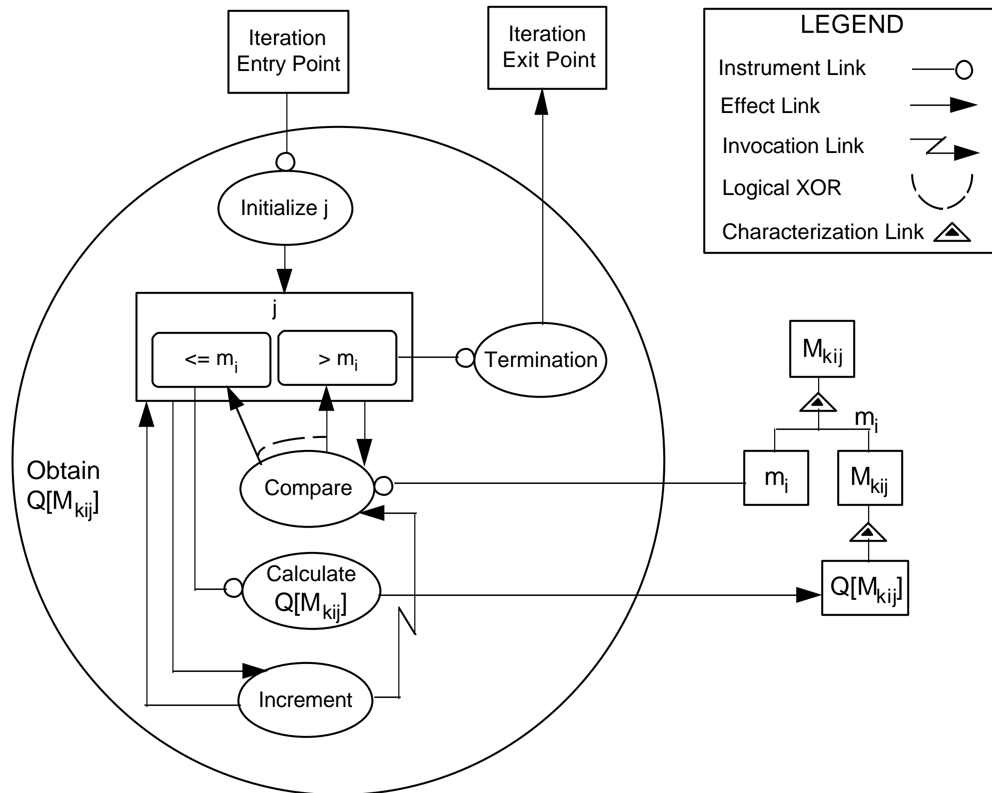

Figure 4: An OPD describing the process of obtaining the Quality Result Q(Mkij) of a Test Sample j of a powder mixture undergoing a Test Series of type Mki. Mki represents a powder mixture, M, undergoing a Test Series i, in a Test Group k. j is a variable used to index the Samples of a Test Series Mki. mi is the number of Samples, Mkij, of tests of type Mki, where $1 \leq j \leq mi$, to be performed.

4.1 A Post-Order Example

The case in point we consider is the process of post-ordering of a binary tree, the C code of which is listed below.

```
post_order (node *t)
{
    if (t==null) return;
    else
    {
        post_order(t->left_child);
        post_order(t->right_child);
        print(t->value);
        return;
    }
}
```

The Object-Process Diagram of the post order process is depicted in Figure 5. To be able to fully describe this process using Object-Process Diagrams, we have augmented OPD's graphic terminology by two additional types of links: the Return Link and the Memory Link. The graphic notation for both links , shown in Figure 5, consist of a lightening symbol above which the letter “R” or “M” is marked to denote the Return and Memory Links, respectively.

Each instance of the recursive process starts at the Recursion Entry Point and ends at the Recursion Exit Point. Both these points appear in Figure 5. The Return Link connects the Recursion Exit Point with a sub-process of the recursive process. Return links emanate only from the Recursion Exit Point. When the Recursion Exit Point is entered, the process does not necessarily end. Rather, using the Return Link, the control returns to the most recent call of the calling process, if such a call exists. Within the calling process, the return is to the subprocess that follows the most recently executed subprocess.

While several Return Links may leave the Recursion Exit Point, at any time during the Post Order (or any other recursive) process execution, exactly one return path is taken. This path is determined by the Memory Link. A Memory Link is a link between two subprocesses of the recursive process that "remembers" (i.e., keeps record of, hence its name) the subprocess executed just before entering the Recursion Entry Point. The Memory Link links that subprocess to the subprocess which should take place after returning from the execution of the called process. When the control arrives at the Recursion Exit Point, it selects the Return Link that leads to the subprocess to which the most recently generated Memory Link also leads. This way the control correctly passes from the Recursion Exit Point to the correct subprocess which to be executed next. The recursive process ends when no more Memory Links are available, implying that the outermost level has been reached.

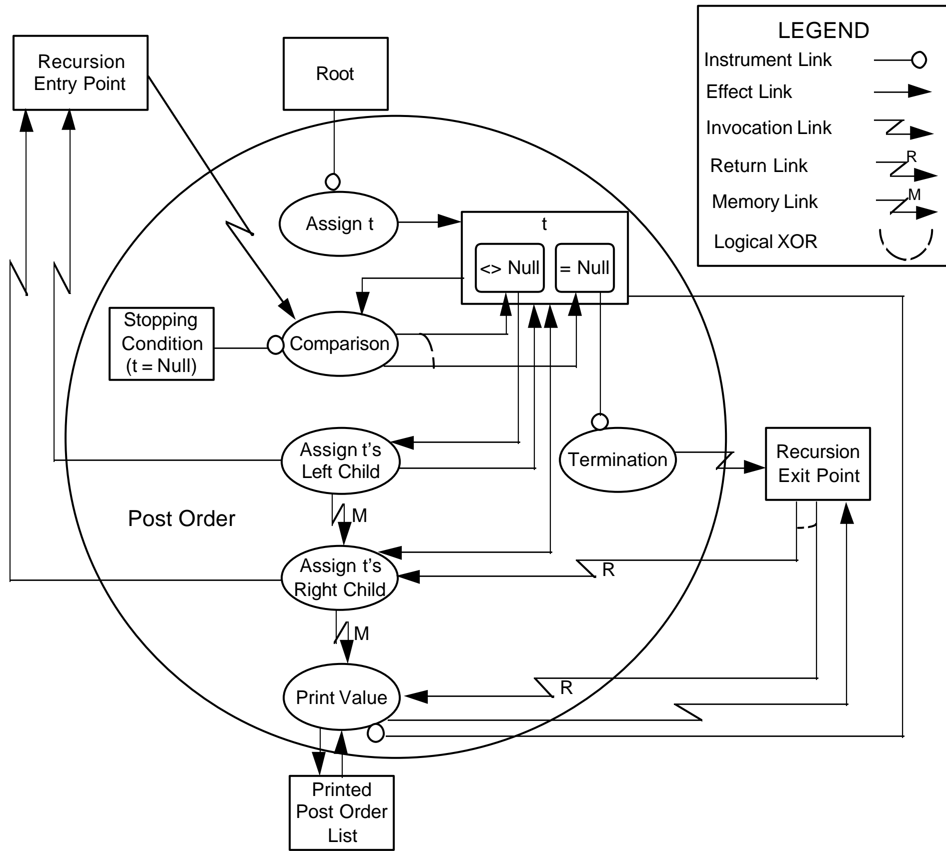

Figure 5: An OPD describing the post ordering process of a binary tree. *t* is the control pointer initially assigned to the root of the binary tree, and during the post-ordering of the tree points to different nodes of the tree. Each node has a right child and a left child, which are null in case the node is a leaf. The process "*Post-order*" prints the values of the nodes of the binary tree in post order.

As noted, the C code of a recursive process is supported by a compiler, which defines and maintains a stack. This stack holds the different values of the calls to the recursive process, along with their environment-arguments and return values. Hence the "visible" code is just a small part of the actual code that executes the recursion. In Object-Process Diagrams that describe recursive processes, the Memory Links are formed in a specific order during the execution of the recursive process. This set of ordered Memory Links is functionally equivalent to the stack of the called instances of the recursive process.

In order to follow the post-order process depicted by the Object-Process Diagram of Figure 5, consider as an example the binary tree of Figure 2(b). Let *t* be the object that traverses the binary tree. At each node *t* is assigned the value (name) of the node it is visiting.. In the first stage, *t* is initialized to point at the root of the tree, namely M. The control enters a branching point, which causes the control to flow to the process *"Assign t's Left Child",* since *t* = M is not null. This results in assigning *t*'s left child, M_1, to *t*. At this point, the first Memory Link is formed between the process *"Assign t's Left Child"* and the process *"Assign t's Right Child"*, and stored in the Memory Link stack shown in Figure 6. The elements of the Memory Link stack are pairs of the form {M_k, (Source→Destination)}, where M_k is the value of the control pointer *t*, and (Source→Destination) is the pair of processes between which the Memory Link passes.

Part (a) of Figure 6 shows the content of the Memory Link stack after the first Memory Link is formed. The Recursion Entry Point is entered again. The Invocation Link, when links a process to an object (or an object to a process) implies the transfer of control to that object (or process), as in a GOTO statement. From the Recursion Entry Point, the control enters the comparison process, where the current value of *t*, which is M_1, is compared to null. This is a recursive call with the parameter M_1. Since $t = M_1$ is not null, the left child, M_{11}, is assigned to *t*. A second Memory Link is formed between the process *"Assign t's Left Child'* and the process *"Assign t's Right Child"*. The stack element {M_1, *Assign t's Left Child'→Assign t's Right Child"*} is pushed into the stack, as shown in Figure 6(b).

The Recursion Entry Point is entered again, and the current value of *t* is compared to null in a recursive call, whose parameter is M_{11}. Since $t = M_{11}$ is still not null, the left child, M_{111}, is assigned to *t*. A third Memory Link, between the process *"Assign t's Left Child"* and the process *"Assign t's Right Child"*, is formed. The stack element {M_{11}, *Assign t's Left Child'—>Assign t's Right Child"*} is pushed into the stack, as shown in part (c) of Figure 6. The Recursion Entry Point is entered again, and *t* is compared to null. Since $t = M_{111}$ is still not null, *t* is next assigned to its left child, which is null. A fourth Memory Link is formed between the process *"Assign t's Left Child'* and

the process *"Assign t's Right Child".* The stack element {M_{111}, *Assign t's Left Child'*—>*Assign t's Right Child*} is pushed into the stack, the current content of which is shown in Figure 6(d).

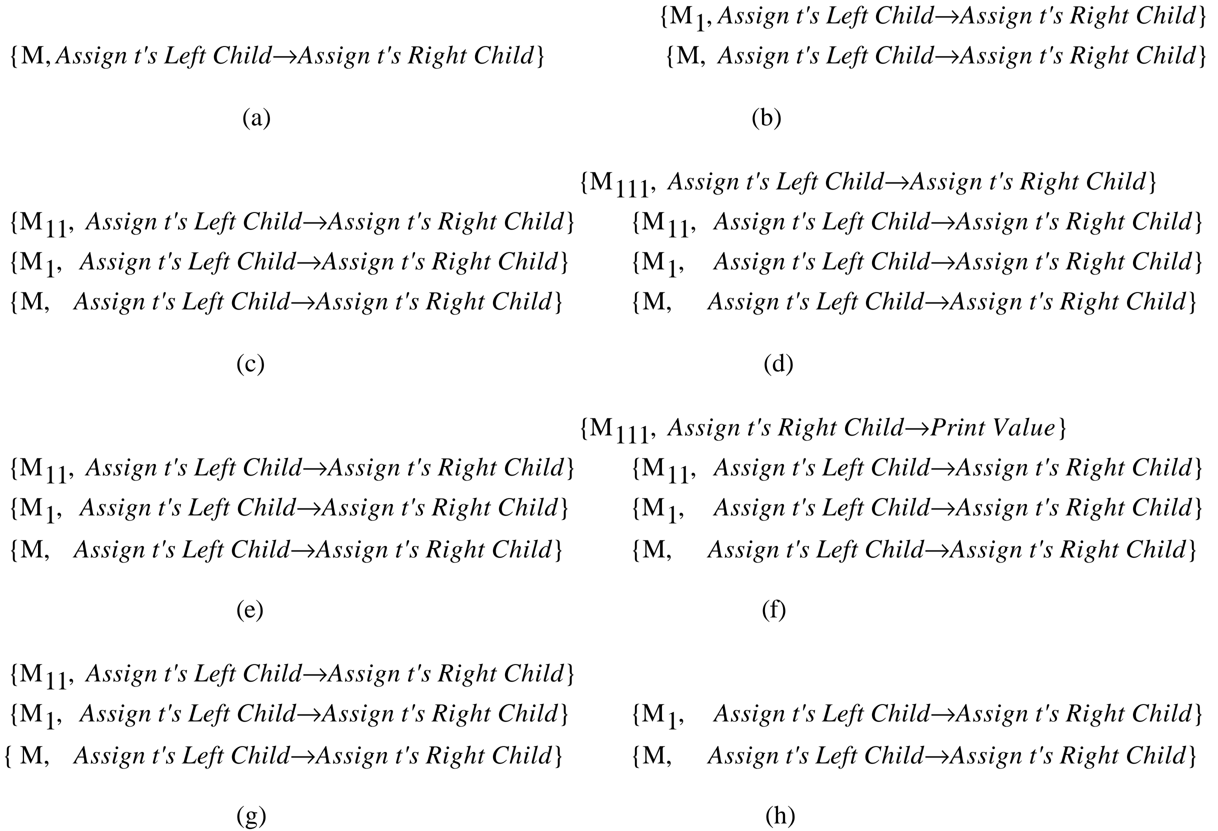

Figure 6: The contents of the Memory Link stack at the various states during the execution of the recursive process *"Post Order"* of Figure 5. Each element in the stack is the pair {M_k, Source→Destination}.

The Recursion Entry Point is entered again and *t* is compared to null. Since this time *t* is null, the Recursion Exit Point is entered for the first time. The control now returns to the subprocess pointed to by the Memory Link emanating from the most recently executed subprocess within the recursion. This subprocess is "*Assign t's Right Child*", which is pointed to by "*Assign t's Left Child*" with *t* being equal to M_{111}. The Return Link consumes the top most Memory Link stack element by a "pop stack" operation, and the state of the stack is shown in Figure 6(e). "*Assign t's Right Child*" assigns to *t* the right child of M_{111}, which is null. A Memory Link between the process *"Assign t's Right Child'* and

the process *"Print Value"* is formed, with t equal to M_{111}, and the corresponding element is pushed into the stack as shown in Figure 6(f).

The Recursion Entry Point is reached again, and since t is null, the Recursion Exit Point is entered. From here the control returns to the level of the calling process—the M_{111} level, where it enters the process pointed to by the Memory Link popped from the top of stack shown in Figure 6(f), to produce the state of the stack shown in Figure 6(g). This process is *"Print Value"*, which finally prints the value of M_{111}. The Recursion Exit Point is reached and the control returns to the level of M_{11}, where it enters the process *"Assign t's Right Child'"*, which is the process pointed to by the Memory Link popped from the top of the stack shown in Figure 6(g). Popping this Memory Link produces the stack whose state is depicted in Figure 6(h). The process *"Assign t's Right Child"* assigns M_{11}'s right child, which is M_{112}, to t. The recursive process ends when the stack is empty, i.e., no more Memory Links are available, in which case the control at the Recursion Exit Point cannot follow any Return Link. At this state the recursion finally terminates and the control flows to the next program construct.

5. THE POWDER MIXTURE QUALITY RECURSION

Having explained the basic recursion mechanism, we now examine the OPD of Figure 7, which describes the recursive process *"Obtain Q(M)"*, of obtaining the quality index of the metal powder mixture. This OPD is a blow-up of the "*Obtain Q(M)*" process depicted in Figure 2(a). The hierarchy of the powder samples, shown on the right hand side of Figure 7, corresponds to the "M tree" in Figure 2(b). M is a metal powder mixture. It is characterized by Q(M) which is the powder mixture's Quality Index, and by m_p Test Groups, M_k, where k = 1,2,...m_p. Each Test Group, M_k, is characterized by $Q(M_k)$ which is the Test Group's Quality Property value, and by m_k Test Series, M_{ki}, where i = 1,2,...m_k. Each Test Series, M_{ki}, is characterized by $Q(M_{ki})$ which is the Test Series' Quality Attribute value, and by m_i Test Samples, M_{kij}, where j = 1,2,...m_i. Each Test Sample, M_{kij}, is characterized by $Q(M_{kij})$ which is the Test Sample's Quality Result.

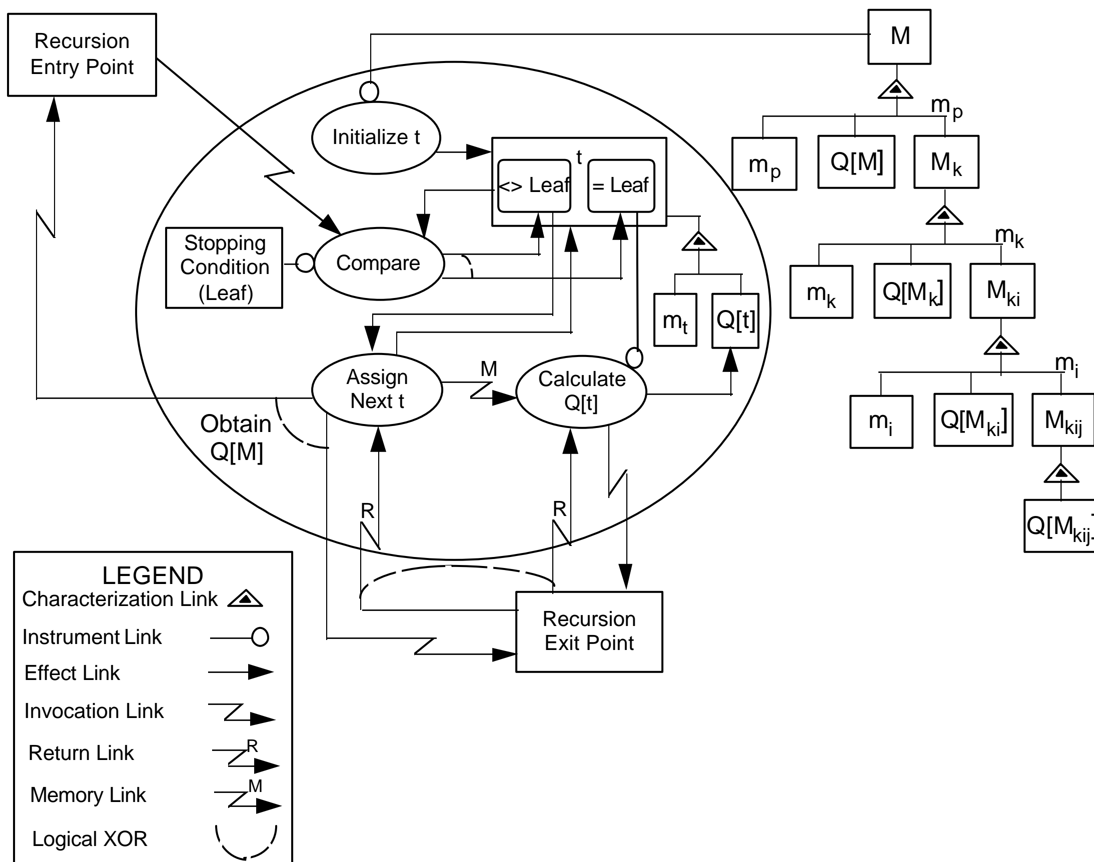

Figure 7: An OPD of obtaining the Quality Index Q(M) of a metal powder mixture M. Q(M) is obtained by recursively visiting modes of the "M tree", shown on the right-hand side of the figure, in a post-order fashion. The nodes' quality results are calculated by the process *"Calculate Q(t)"*.

The recursive process "*Obtain Q(M)"* computes the quality index using post ordering of the M tree. *t* is the pointer to the current node in the tree. When "*Obtain Q(M)*" is called with a node that has no children, then an actual quality test Sample is carried out, the quality result of that node is calculated, and the control enters the Recursion Exit Point. If the current node is not a leaf, i.e., it has children, then the control enters the loop "*Assign Next t*". This loop attempts to assign to *t* its next child. If no more unvisited children remained, the control enters the Recursion Exit Point. Otherwise, *t* is assigned to its next child, the control enters the Recursion Entry Point, and another instance of the

process "*Obtain Q(M)*" begins. This iteration is repeated until all of *t*'s children have been assigned to *t*. As noted, the Recursion Exit Point is entered either following the calculation of the quality result of a leaf node, or when the process *"Assign Next t"* "discovers" that *t* has no more children that have not been visited, by noting that $x > m_t$.

When the control is at the Recursion Exit Point, it follows one of the return paths to return to the instance of the calling process. The return path is determined by the most recently formed Memory Link. One Memory Link, shown in Figure 7, connects the process *"Assign Next t"* to the process *"Calculate Q(t)"*. The second Memory Link, shown in Figure 8, is within the process *"Assign Next t"*. The formation of both Memory Links is determined by the process *"Assign Next t"*. If the Recursion Exit Point is entered following the occurrence of the process *"Calculate Q(t)"*, the return path is to the process *"Assign Next t*". Otherwise, the Recursion Exit Point was reached from the process *"Assign Next t"* when all of *t*'s children have been visited, that is, all their quality results have already been determined. In this case, the return path is to the process *"Calculate Q(t)"*. The process "*Obtain Q(M)*" is repeated until termination of all the recursive calls that took place.

To fully understand the entire recursive process of obtaining the total quality result of the powder mixture, we follow the details of the process execution. Figure 8 shows an OPD of the process "*Obtain Q(M)*" with the subprocess *"Assign Next t"* blown up. On entering the process "*Obtain Q(M)*", *t* is initialized to M. The process "*Compare"* compares *t* to a leaf. Next, the control flow enters a branching point which passes the control to the process "*Assign Next t*", since t = M is not a leaf. This results in initializing x, the internal counter of the loop, to 1. A second branching point is reached. Since $m_t = m_p$ is equal to 2, there is a call to the process "*Assign t's Lowest* Index". This process assigns to *t* different values according to the current value of *t*. if $t = M$ then M_x is assigned to *t*. if $t = M_k$ then M_{kx} is assigned to *t*. if $t = M_{ki}$ then M_{kix} is assigned to *t*. Since $t = M$ and $x = 1$, the process "*Assign t's Lowest* Index" assigns M_1 to *t*. The first Memory Link is formed between the process *"Assign t's Lowest Index"* and the process *"Increment"*. The stack element {M, *Assign t's Lowest Index*→ *Increment*} is pushed into the Memory Link stack shown in part (a) of Figure 9.

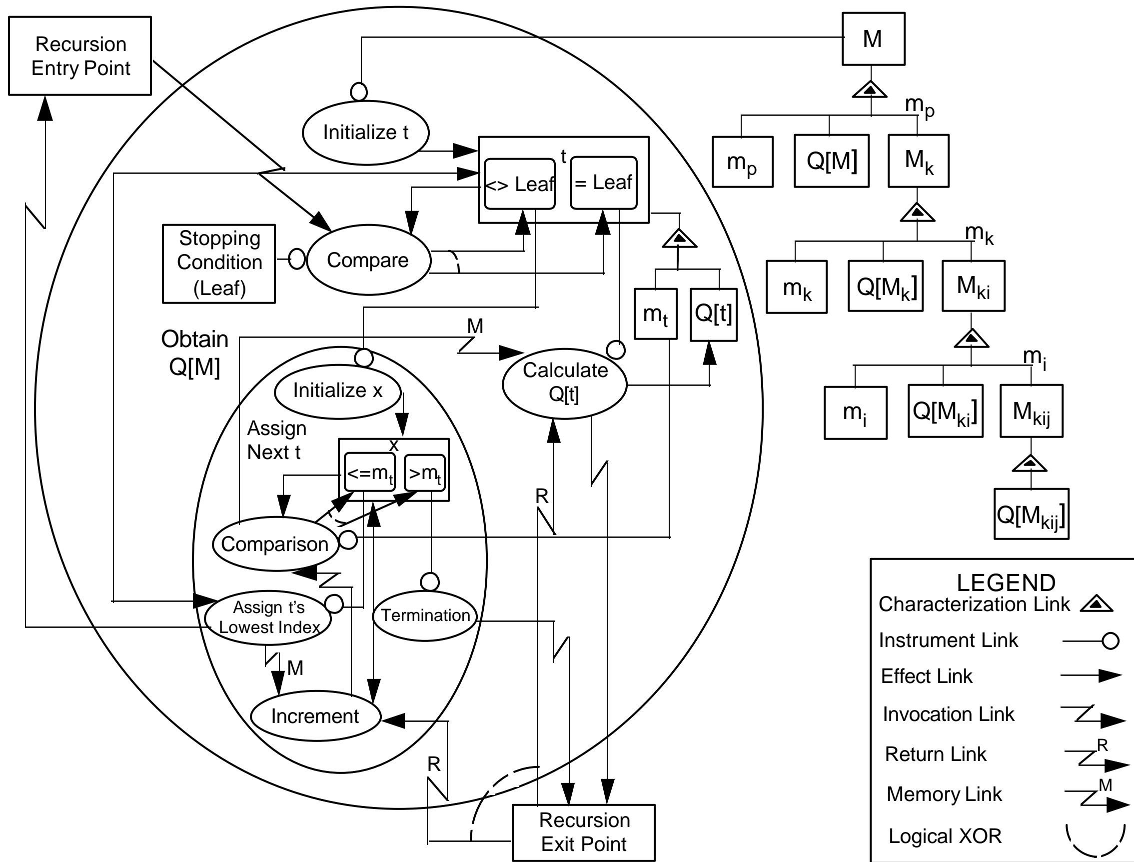

Figure 8: An Object-Process Diagram (OPD) which describes the process of obtaining the Total Quality Result Q(M) of a powder mixture M. This figure is similar to Figure 7, but it includes a "blow-up" of the process "*Assign Next t*" which is a loop that assigns to *t* the next test type index that should be determined, using a post-order sequencing of the M tree. x is the local counter of the loop.

The Recursion Entry Point is entered again. This is a recursive call to "*Obtain Q(M)*" with the parameter M_1. $t = M_1$ is still not a leaf, so another local variable named x is initialized to 1, and since it is less than $m_t = m_k$, which is 2, t is assigned to M_{11} by the process "*Assign t's Lowest* Index". The second Memory Link is formed between the process *"Assign t's Lowest Index"* and the process *"Increment"*. The stack element {M_1, *Assign t's Lowest Index*→ *Increment*} is pushed into the Memory Link stack shown in part (b) of Figure 9.

The Recursion Entry Point is entered again. This is a recursive call to "*Obtain Q(M)*" with the parameter M_{11}. $t = M_{11}$ is still not a leaf, so another local variable x is initialized to 1, and since it is

less than $m_t = m_i$, which is 2, t is assigned to M_{111}. The third Memory Link is formed between the process *"Assign t's Lowest Index"* and the process *"Increment"*. The stack element {M_{11}, *Assign t's Lowest Index→ Increment*} is pushed into the Memory Link stack shown in Figure 9(c).

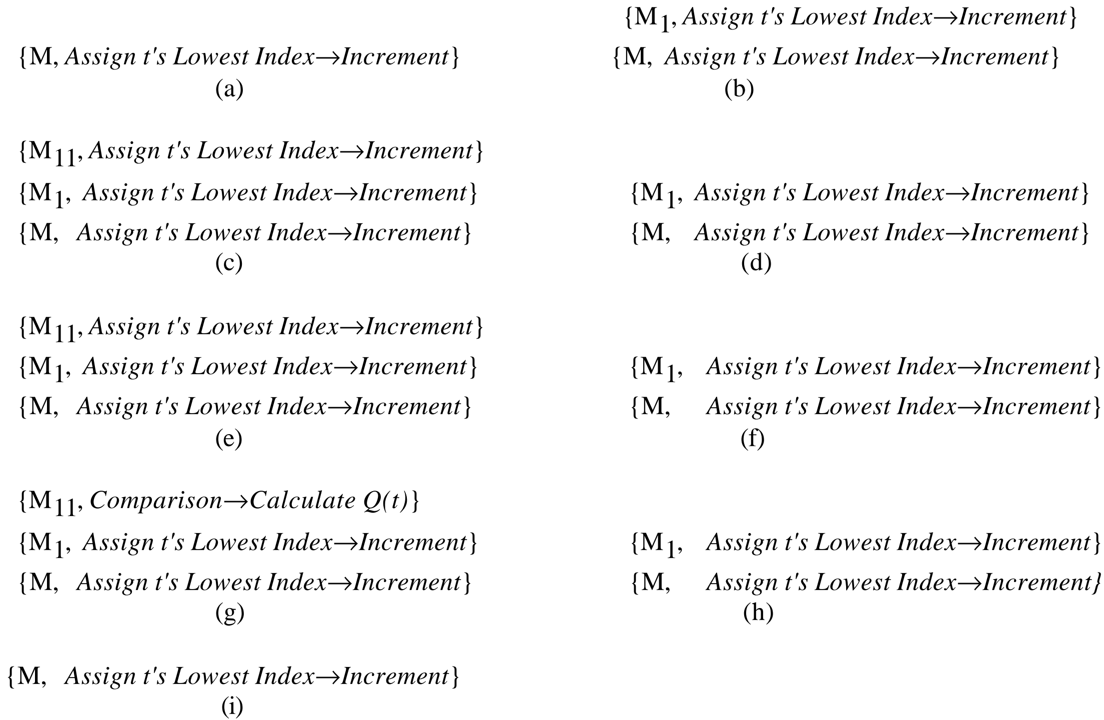

Figure 9: The contents of the Memory Link stack at the various states during the execution of the recursive process *"Obtain Q(M)"* of Figure 8. Each element in the stack is the pair {M_k, Source→Destination}.

The Recursion Entry Point is entered again, followed by a recursive call to *"Obtain Q(M)"* with the parameter M_{111}. This time $t = M_{111}$ is a leaf, so $Q(M_{111})$ is determined. Next, the Recursion Exit Point is reached. The top Memory Link, which connects the processes "*Assign t's Lowest Index*" and *"Increment"* is popped from the top of the stack as shown in Figure 9(d). This Memory Link is used along with the Return Link from the Recursion Exit Point, and the control returns to the level of M_{11}, where it enters the process of incrementing x of the third loop to 2. Since x is now equal to $m_t = m_i$ and $t = M_{11}$, the process *"Assign t's Lowest Index"* assigns M_{112} to t. Another Memory Link is formed between the process *"Assign t's Lowest Index"* and the process *"Increment"*. The stack element {M_{11}, *Assign t's Lowest Index→ Increment*} is pushed into the Memory Link stack shown in Figure 9(e).

The control enters the Recursion Entry Point again. This is a recursive call to "*Obtain Q(M)"* with the parameter M_{112}. $t = M_{112}$ is a leaf, so $Q(M_{112})$ is determined. Next, the control enters the Recursion Exit Point. The top Memory Link from the process "*Assign t's Lowest Index*" to the process "*Increment"* is popped from the stack, as shown in Figure 9(f). This Memory Link is used along with the Return Link from the Recursion Exit Point. The control returns to the level of M_{11}, where it enters the process of incrementing x of the third loop to 3. x is now greater than $m_t = m_i$, so another Memory Link is formed between the process *"Comparison"* and the process *"Calculate Q(t)"*. The stack element {M_{11}, *Comparison* $\rightarrow$ *Calculate Q(t)*} is pushed into the Memory Link stack shown in Figure 9(g).

Since x is now greater than $m_t = m_i$, the control resides in the state equal to the stopping condition of the third loop. The termination process is executed, and the Recursion Exit Point is entered. The top Memory Link is popped from the stack, as shown in Figure 9(h). The Return Link, simultaneously with the popped Memory Link from the process *"Comparison"* to the process *"Calculate Q(t)"*, directs the control to carry on the process of calculating $Q(M_{11})$. The Recursion Exit Point is reached, and a Memory Link is popped from the stack, as shown in Figure 9i. This Memory Link returns the control to the level of M_1, where it enters the process of incrementing x of the second loop to 2. Since x is now equal to $m_t = m_k$, and $t = M_1$, the process "*Assign t's Lowest Index*" assigns M_{12} to t. The process is repeated until Q(M) is obtained and the recursive process terminates since no more Memory Links are available.

6. DISCUSSION

In this paper we have shown how control flow constructs, including composition of statements, conditional or branching statements, iteration, and recursion can be represented in Object-Process Diagrams. All of these control flow constructs are instrumental not only for devising algorithms, but also for analyzing real-life systems. It may therefore be useful and important to be able to describe them graphically in an Analysis/Design methodology. Although most analysis and design

methodologies have a means of representing composition of statements and branching, they are not well suited for representing iteration and recursion. In this work we have augmented OPD's graphic vocabulary in order for it to also be able to represent iteration and recursion. The graphical representation of iteration is rather straightforward. As for recursion, although we have shown it to be possible to be represented graphically, the question remains whether this method of specification is preferable to pseudo-code. As is often the case with methods we are more familiar with, writing pseudo-code for a recursive process may seem easier then describing the same process using an OPD. But when accustomed to OPDs, representing recursion should not be problematic. What should not be overlooked is the fact that behind both a pseudo-code specification and OPD specification of a recursive process there is a stack mechanism that "remembers" (in a LIFO order) all the calls of the recursive process that took place.

Statements, branching, loops, and recursion are principal building blocks of algorithms, and we have shown their representation by OPDs. This implies that OPDs can be used for representing many sorts of algorithms. OPDs can therefore be used to start a top-level specification and recursively scale it up by adding more details until, at the most detailed level, only simple (atomic) processes and objects are represented. The complete OPD set provides a full specification of the system under investigation and design.

In this paper we have not discussed recursive processes that return **values** from an instance of the recursive process to the instance which called it. An example of such a process is a recursive process which counts the number of nodes in a binary tree. Representing this kind of recursion in OPDs can be achieved by adding a counting process, but the exact details require further research.

REFERENCES

[1] Aho, A. V.,, Hopcroft, J. E., and Ullman, J. D., *Data Structures and Algorithms*, Addison-Wesley, Englewood Cliffs, NJ, 1987.

[2] Sethi, R., Programming Languages: Concepts and Constructs, Addison-Wesley, 1989.

[3] Coad and Yourdon, Object Oriented Analysis, Prentice-Hall Englewood Cliffs NJ, 1991.

[4] Rumbaugh, J., Blaha, M., Premerlarni, W., Eddy, F., and Lorenson, W., Object-Oriented Modeling and Design, Prentice-Hall Englewood Cliffs NJ, 1991.

[5] Jacobson, I., Christersson, M., Jonsson, P., and Overgaard, G.G., Object-Oriented Software Engineering, Addison-Wesley, Reading MA, 1992.

[6] Shlaer, S., and Mellor, S., Object LifeCycles: Modeling the World in States, Prentice-Hall Englewood Cliffs NJ, 1992.

[7] Booch, G., Object-Oriented Development, IEEE Transactions on Software Engineering, February 1986, pp. 211-221.

[8] Harel, D., Statecharts: a Visual Formalism for Complex Systems, Sci. Comput. Program, Vol. 8, pp. 231-274, 1987.

[9] Embley, D. W., Kurtz, B.D., and Woodfield, S.N., Object-Oriented System Analysis: a Model-Driven Approach, Prentice-Hall Englewood Cliffs NJ, 1992.

[10] Booch, G., Rumbaugh, J., and Jacobson, I., The Unified Modeling Language for Object-Oriented Development, Version 0.9, Rational Software Corporation, July 1996.

[11] Firesmith, D., Henderson-Sellers, B., and Graham, I., Open Modeling Language, Version 0.1, The OPEN Consortium, July 1996.

[12] Jackson, M., System Development, Prentice-Hall International, 1983.

[13] Dori, D. Object-Process Analysis: Maintaining the Balance Between System Structure and Behavior, Journal of Logic and Computation. Vol. 5 no. 2 pp. 227-249, 1995.

[14] Dori, D. and Goodman M. On Bridging the Analysis-Design and Structure-Behavior Grand Canyons with Object Paradigms. Report on Object Analysis and Design, Vol. 2 No. 5 pp. 25-35, 1996.

[15] Dori, D. and Goodman M. From Object-Process Analysis to Object-Process Design. Annals of Software Engineering, Vol. 9, pp. 1-25, 1996.

[16] Dori, D. Object-Process Analysis of Computer Integrated Manufacturing Documentation and Inspection Functions. International Journal of Computer Integrated Manufacturing, Vol. 9, No. 5, pp. 339-353, 1996.