Determining Granularity of Independent Tasks for Reengineering a Legacy System into an OO System

Richard Millham, Martin Ward and Hongji Yang
De Montfort University, England
Richard.Millham@shaw.ca & mward, hyang@dmu.ac.uk

Abstract *Reengineering a COBOL legacy system is a difficult multi-step process, particularly when the COBOL legacy system is a sequential procedural-driven system which is being reengineered into an object oriented, event-driven system. In this scenario, it is necessary to analyse the legacy system in order to identify which tasks can be executed independently and which tasks must be executed sequentially. The focus of reengineering is too often based on theory rather than based on experience gained from real-world examples. This paper hopes to address this imbalance by providing a practical application of reengineering to an actual legacy telecommunications system.*

Keywords: Wide Spectrum Language (WSL), Unified Modelling Language (UML), Reverse Engineering, COBOL Legacy Systems, Reengineering

1. Introduction

Throughout the world, there are many legacy systems, written in COBOL, which were designed many years ago but still function today. There are many reasons for their continued operation but these reasons include the high cost of developing a new system, the fact that these systems fulfill critical business functions, and the fact that the business cannot afford the downtime that switching to and testing of a new system would involve.

The example legacy system that is used in this paper constitutes one of the core business systems used by Telus, a Canadian communications company. These legacy systems were developed during the 1960's and 1970's as procedurally-driven, mainframe-based programs. Maintenance changes over the years obfuscated the original system's design and made it difficult to modify the systems to handle new user requirements, such as data services. Developing new systems to replace the existing systems was hampered by the high cost of new system development and by the real threat to billing deadlines posed by system downtime during the switchover. The only feasible alternative to this dilemma is to re-engineer the legacy system into a more modern representation, such as an event-driven, object-oriented system. An object-oriented system, with any code changes limited to its encapsulating object, promises fewer side effects and lower maintenance costs than the global-scoped and globally impacting COBOL variables.

The first step in transforming a legacy procedural system into an object oriented system is to identify the set of objects and their associations. After objects and their associations have been identified, the next step is to determine which tasks can be executed independently [9].
One way to do this is to model these tasks in the UML activity diagram as activities. An UML activity diagram represents the dynamic view of a system as a sequence of steps grouped sequentially as parallel control branches. An activity diagram is similar to a statechart except that a statechart distinguishes between states, activities, and events.

Non-independent tasks will be modelled as sequential activities and independent tasks will be modelled as parallel activities. The definition of parallel processes, in our paper, is a set of processes which can be executed in any order. Synchronisation bars are used to synchronise the divergence of sequential activities into parallel tasks or the merging of parallel tasks to a sequential task. These enable the control flow to transition to several parallel activities simultaneously and to ensure that all parallel tasks complete before proceeding to execute the next sequential task.

This next step after determination of task independence is to identify pseudo-events from the legacy source code and to determine the degree of asynchronicity of each of these pseudo-events. Pseudo-events may be procedure calls, I/O operations, error raising, or system interrupts. These pseudo-events are then represented in a sequence diagram as message passing between objects. An example, if a method of object A calls another method of object B, this method/procedure call is represented as a call event; this call event is represented as a call message between objects A and B in a sequence diagram. To model a message in a sequence diagram, this message must be represented as asynchronous or synchronous. In order to determine this message's asynchronicity, it is necessary to determine whether the task incorporating this method call and its immediately succeeding tasks can execute independently or whether they must execute sequentially. If the task incorporating the method call cannot execute independently from its immediately successive task, this message is modelled as a synchronous message; otherwise, this message is modelled as an asynchronous message.

This independent task analysis should be seen as one step in a difficult multi-step process of reengineering a COBOL legacy system. We adopt the approach of develop tools and an intermediate language based on making use of existing tools and existing languages; we use the Wide Spectrum Language (WSL) [8, 14] for intermediate language representation and the Unified Modelling Language (UML) for documentation and code generation. Our approach is to reverse engineer the source COBOL code into a WSL representation that is then transformed into a UML format. This UML format can then be exported to other UML tools for documentation and re-development purposes.

2. Related Work

In his work with reengineering COBOL legacy systems, Tsai [13] concludes that many COBOL programs are very difficult and expensive to maintain and consequently identifies the need to re-engineer these legacy systems, often to an object-oriented system. Tsai states that converting a legacy system to an object-oriented system is difficult because the programming style that was used when the legacy system was developed does not incorporate itself into the object-oriented paradigm well. Tsai identifies the use of global variables within COBOL as a major stumbling block [13]. Another stumbling block which we have identified, and which is addressed in this paper, is the assumption of purely sequential processing in these legacy systems. Consequently, no independent tasks have been identified within this legacy system, and this makes it difficult to convert to a multi-processing, event-driven, object-oriented paradigm.

Our method of using procedures as the maximum granular unit is similar to Gall's identification of procedures as incorporating the highest level of system control [5]. We use a static call analysis to our legacy system in order to identify dependencies between procedures [4].

Lam tracks the amount of parallelism and misprediction of branching that is inherent in various programs of various types, such as numeric or database programs. The degree of parallelism that was detected was discovered to depend, in part, on the type of programs. Database and numerical programs tend to have a higher degree of parallelism than business-oriented programs, such as our legacy system [7].

A typical approach to reengineering of procedural COBOL code to an object-oriented architecture [15] is to partition programs into abstract data types and reallocate procedural programs according to objects of processing. The result is a set of classes in an object-oriented COBOL. Apparently obtaining just classes in migration has limited application in a large system. Other unsolved issues with this work include how to produce non-redundant classes and how to get the obtained classes in the new system to communicate and collaborate with each other.

Levey [16] illustrated how older methods of writing COBOL programs lead to logic that clogs programs with complexity and chokes off further development; and by using objects, a methodology for taking these older systems and rebuilding them is described. In the process, the programmer gradually removes the older logic and replaces it with object oriented code. The result is systems that are adaptable to new technologies in COBOL, even though it is not an object oriented language. It seems that the described approach employs many manual operations, which may not be effective in dealing with medium to large system.

In [18], a clustering/grouping method is used to identify objects and the algorithms based on this method are implemented in the Maintainer's Assistant, a semi-automatic reverse engineering tool. Grouping can start with either a variable or a program statement that uses a variable, and then program transformations are employed to "regroup" a closure of variables and operations on these variables to form an "object" This technique is only an early step towards migrating a data intensive program into an object oriented program.

[17] uses both cluster and concept analysis for object identification, combining legacy data structures with legacy functionality. This approach, as in [18], takes the view that records are a natural starting point, and that decomposing the records into smaller ones is necessary, and a method of doing so was proposed. Concept analysis can help with problems that clustering analysis cannot solve, such as placing one item in different partitions and missing out useful items from partitions. Still, the results of this analysis results lack information needed for building UML diagrams.

Issues on identifying the optimal granular unit for independent tasks are about factors such as the size of the granular unit, their dependencies with other units, the communication overhead of independently-execution tasks and the optimisation of available processors. Our work focuses on the size of the granular unit and the dependencies among them. Because our work focuses on a platform-independent architecture, we will therefore not address the factors of determining the optimal granular task size to minimise communication overhead among processors and to optimise use of available processors is highly dependent on the architecture of the supporting hardware platform.

3. Our Investigation

We first convert the COBOL legacy system into WSL using a set of COBOL to WSL conversion rules that were formulated by Ward [14], which defined the basis of the Wide Spectrum Language, and Liu [8], which extended WSL to encompass other common programming constructs. Then this WSL representation must, in turn, be analysed in order to determine independent and non-independent tasks.

The following algorithms, and an underlying tool, were developed to determine independence of tasks within WSL rather than a particular programming language for a number of reasons. Because the number of programming languages is large and even within a particular programming language, such as COBOL, there are many dialects, developing a tool to analyse each particular programming language would be a huge undertaking. Instead, we chose to develop a tool that could analyse tasks in an intermediate language and then write converters to convert the source code written in a particular programming language to this intermediate language. This intermediate language, WSL, was chosen, because as a wide spectrum language, WSL can represent a wide range of programming languages, whether it is a high-level programming language, such as COBOL, or a low-level one, such as an assembly language.

WSL has other advantages as well. It has excellent tool support, in the FermaT transformation system [19] which allows transformations and code simplification to be carried out automatically. It has the capability of enabling proof-of-correctness testing. Another important reason why WSL was chosen was that it gives programming language independence to the evaluation of task independence. WSL was also specifically designed to be easy to analyse and transform.

Another advantage of WSL is that an important concept, codeline, used in the study, can be unified. When evaluating task independence using the task granularity of individual codelines, the concept of a codeline differs greatly among programming languages. A high-level programming language's codeline can perform the same task as multiple lines of assembly code. Thus, the degree of task independence using the granularity of individual lines of code in a high-level programming language like COBOL may not be necessarily similar to the task independence using the granularity of individual lines of code in a low level programming language like assembly language. However, if the source code has been converted to WSL and the evaluation of task independence is performed on this WSL code, an individual codeline remains the same regardless of whether the source code was COBOL, assembly, or FORTRAN.

The ultimate goal of our research is to reengineer legacy source code from a sequential, procedural-driven system to that of an object-oriented, event-driven system and then represent this transformed system in UML, with the capability of CC++ code generation. CC++ is an extension of C++, proposed by K Chandy and C Kesselman, designed to operate on a multiple-instruction, shared memory platform. [20]. Using a set of programming primitive constructs, a programmer is able to explicitly indicate which tasks can be executed in parallel and which variables are to be used to synchronise the execution of parallel-executing to their successive sequential tasks.

The subset of WSL that we will be using consists of IF and WHILE. The COBOL legacy source code has been restructured to transform any existing GOTOs into IF and WHILE constructs before the COBOL's conversion to WSL [22].

Our investigation involves determining what level of granularity of tasks (procedural, programming block, or individual code line) is the best for determining task independence. Our approach consists of four steps:

1. Determining Data and Control Dependencies – determining whether two tasks T1 and T2 are independent involves identifying whether any control or data dependencies exist between these two tasks. A dependency can be defined where one unit of execution, or task, T1 is dependent on another task T2 for its execution. These dependencies may be a control dependency where the expressions, and their constituent variables, of the control constructs enclosing the task determine if and how this task will be executed. The dependencies may also be a data dependency where two tasks, T1 and T2, share common variables and where the correct computation of results in task T2 is dependent on the preceding computation of results using shared variables in task T1.
2. Program Block Identification – A program block is a logical unit of execution and it represents an intermediate level of possible explicit granularity in this legacy system. A procedure will contain one or more programming blocks and a programming block contains one or more code lines.
3. Individual Codeline Evaluation – a code line, or statement of legacy code, represents the minimal granular unit possible in this legacy system. An individual code line represents the lower bound of possible explicit granularity in this legacy system.
4. Procedure Granularity – a procedure, in a procedurally-designed system, is one of the largest self-contained granular units possible. According to procedural design theory which was prominent in programming during the time that this legacy system was developed, a procedure should encapsulate one or more related functions along with their associated data, in the form of locally-used variables. By assessing tasks at the procedure granularity level, we can take advantage of the granularity present in the explicit modularisation of procedural design. A procedure represents the upper bound of possible explicit granularity in this legacy system.

The detailed techniques can be seen in the remaining subsections.

3.1 Determining Data and Control Dependencies

All levels of granularity use a common function Check_For_Shared_Vars which has two parameters, one set of codeline(s) and another set of codeline(s). If one set of codeline(s) updates one or more variables that are currently being used with the other set of codeline(s), then the function returns true; otherwise, the function returns false.

3.2 Program Block Identification

For program block analysis, we must first identify and categorise the program blocks and then determine the task independence of each program block. This determination is accomplished by using the following algorithm:

1) *within each procedure, determine the programming blocks. Programming blocks are delimited by keywords of the if-then or while-do constructs. In the case of nested programming blocks (1..n), each programming block is assigned a nesting level with 1 begin the outermost programming block and n being the innermost programming block. In order to distinguish between different programming blocks of the same procedure at the same level, each programming block of the same level is assigned a sequence of 1..n where 1 is the first programming block encountered and n is the last programming block encountered. In the case of procedures with no control logic, all lines of code within the procedure are grouped into a single programming block.*
2) *after all the programming blocks have been identified, determine, for each procedure, which programming block level is the most numerous. This most numerous programming block level determines, in the case of nested programming blocks, how each code line will be evaluated. All code lines belonging to a programming block of this most numerous level are evaluated as part of this programming block, regardless if they belong to other programming blocks as well. For code lines that fall outside this particular programming block level, these code lines are assigned their most innermost programming block.*

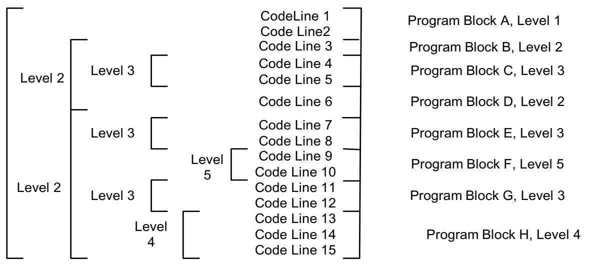

Figure 1. Code Lines and their Levels

The algorithm to determine which programming blocks are independent is as follows:

```
1)for each procedure, initialise the task sequence number to zero;
2)for each pair of program blocks within current procedure:
    a. if Check_For_Shared_Vars(codeline(s) of first program block, code
       line(s) of second program block) returns true, then update code line(s)
       of first program block to the task sequence number; increment to next
       task sequence number; and update code line(s) of next program block
       to the task sequence number,
```

b.otherwise {no shared variables} update code line(s) of first and second program block to same task sequence number since they both can execute in parallel.

Each pair of program blocks are managed so that the second program block of the current pair comes the first program block of the next pair.

3.3 Individual Code line Evaluation

The algorithm to check individual programming codelines is as follows:

```
1)For every procedure
current task sequence number is initialised to 0;
For every code line (curcodeline) of current procedure
If curcodeline or next code line is a While or if construct, then update
   curcodeline's task sequence number to current task sequence number;
   increment to the next current task sequence number; update next code
   line's task sequence number to current task sequence number;
   Otherwise,
If Check_For_Shared_Variables(curcodeline, next code line) returns true,
   then these two tasks are not independent tasks so update next code line to
   next task sequence number;
Otherwise, {no shared variables so these two tasks are independent};
Update curcodeline's task sequence number to current task sequence
   number;Update next code line's task sequence number to current task
   sequence number.
```

Each code line is evaluated in context of its procedure rather than in context of its program. The reason for this evaluation is that procedures are a logical entity to encapsulate code lines and using this method, each code line is evaluated once, within the context of its procedure. If the code line was evaluated on a program-level basis and in relation to the procedural call graph, a code line may be evaluated several times.

If a control construct is encountered in a code line, the task sequence number is incremented to the next task sequence number. The reason for this incrementation is that any subsequent code lines that are enclosed by this control construct can not be evaluated in parallel with the control construct code line. The control construct code line must be evaluated first and then any enclosed code lines can be evaluated.

The maximum number of program blocks of the same level is selected in order to maximise the number of granular units that can be considered for parallelisation within a procedure.

3.4 Procedure Granularity

The algorithm to determine task independence among procedures is as follows:

```
1.Determine the main controller function(s), through analysis of fan-in and
   fan-out of procedures. Controller functions typically have a high fan-out
   (they call many other procedures) but have a very low fan-in (they are
   called by very few other procedures). In this case, the COBOL->WSL
   rules stipulate that the main controller function for a COBOL source file,
   whether COBOL program file or copybook, is created with the
   procedure name <SOURCE-FILENAME>_VAR-INIT;
2.The main controller function is given a procedure sequence number of 1
   and assigned to a temporary table, TmpProcsAlreadyAssigned;
3.Increment the procedure sequence number variable, ProcSeqNo, by one
   to two;
4.All of the procedures that are called by the main controller function are
   then put into a temporary table, TmpDetermineParallelProcs, and then
   analysed for shared variables;
5.While updating_procedure_sequence_number is true
if one or more procedures, procsA, from the TmpDetermineParallelProcs
shares no variables with any other procedure in the
TmpDetermineParallelProcs
   a)set procedures', procsA, sequence number to ProcSeqNo
   b)assign procedures, ProcsA, to table TmpProcsAlreadyAssigned
   c)assign all procedures that are called from procedures, ProcsA, to
     table TmpDetermineParalleProcs
   d)delete procedures, procA, from table TmpDetermineParallelProcs
     now that their procedure sequence number has been assigned
   e)procedure_sequence_number is incremented by one
   f)set updating_procedure_sequence_number to true
   otherwise, set updating_procedure_sequence_number to false;
6.The While Loop continues until one of the three conditions are met
   a.The number of procedures to be analysed for parallelism is too small
      to be analysed – one remaining procedure
   b.No updating of procedure task sequence numbers have been done and
      no procedures have been deleted from table,
      TmpDetermineParallelProcs;
7.The remaining procedures that have not been assigned a procedure
   sequence number now have this number assigned as a sequence number,
   incremented by one for each procedure. An example, the first remaining
   procedure is assigned for its procedure sequence number the
   procedure_sequence_number + 1, the second remaining procedure is
   assigned procedure_sequence_number + 2.
```

4. Experiments

4.1 Procedural Level Granularity

When examining the task sequence numbers of each individual procedure, we find that of the 37 total procedures, one procedure is deemed to be the main controller procedure and is assigned a task sequence number of one. Four procedures, which are called from this main controller procedure, can execute concurrently and consequently, are assigned task sequence number of two. Of the remaining 32 procedures, these procedures are assigned task sequence numbers sequentially (3 onward).

On the procedural level, there are on average 75 lines of codelines per parallel procedure. For a non-parallel procedure, the average is 28.8 lines. Following the logic that the fewer codelines that a procedure has, the less chance of a data or control flow conflict occurring with another procedure, this discovery that the average number of code lines for a parallel procedure is much larger than that of a non-parallel procedure seems puzzling and warrants further investigation. However, when we examined the parallel procedures, we noticed that these procedures were either initialisation or termination routines (which have large number of code lines but have only functional cohesion – the code lines in the procedure share a common function of initialisation/termination but do not have anything else in common, hence these code lines have less chance of sharing variables with each other) or main input-output routines (ex: Write-Trans-Master). The main input-output routines have a large number of codelines because of the large number of record fields involved in these routines which must be represented as operations in code lines. However, these codelines are limited to mostly input/output operations; hence, they lack the complex logic or common variables which a smaller, non-parallel procedure might have.

From the results of the experiment at the procedural level granularity, it seems that the procedures at the same calling level from the main calling procedure(s), for the majority of cases, share variables. An inspection of these variables reveals that often these shared variables are exclusive error variables used in error logging. Although these variables are shared, the values of these variables are merely logged and are not used in the calculation of further values.

One method to reduce these shared variables is to separate these error logging variables from further consideration. Variable contention among error logging variables could be handled with a special queued error logging monitor mechanism.

4.2 Program Block Level Granularity

Of the 37 procedures, all the programming blocks in 30 procedures have a task sequence number of zero (they can execute independently without requiring any preceding programming blocks to execute beforehand). The remaining seven procedures have the following statistics:

Procedure Number	Number of Tasks	Number of Task Sequences within Procedures	Number of Independent Tasks at Task Sequence Zero
1698	48	32	6
1705	19	11	9
1718	27	13	15
1720	136	69	69
1721	54	39	16
1722	59	49	11
1726	71	49	23

On the programming block level, the average number of code lines per parallel programming block is 28.9 code lines. The average number of codelines per non-parallel programming block is 5.6 code lines. Again, following the logic that the fewer code lines that a programming block has, the less chance of a data or control flow conflict occurring with another programming block, this discovery that the average number of code lines for a parallel programming block is much larger than that of a non-parallel programming block seems puzzling and warrants further investigation. When we investigated further, we found that the most common number of code lines per non-parallel programming block is one (it occurs 76% of the time) – according to the programming block definition algorithm, a one code line programming block occurs if this code line would have a dataflow or control flow conflict with the succeeding code line; if no conflict occurs, the succeeding code line aggregates with the current code line to form a programming block until a code line is encountered which conflicts with existing code lines in a programming block. It is only then that the current programming block is defined (minus the conflicting code line) and a new programming block is formed with the conflicting code line. With this algorithm, the number of code lines in a non-parallel programming block is kept to a minimum in order to increase the number of code lines within a parallel programming block.

4.3 Individual Code Line Level Granularity

Individual code line granularity seems to produce the largest number of independent tasks per unit. However, at the same time, it also produces the largest disparity in the number of independent tasks detected per unit. An example, the number of independent tasks per unit can be as high as 99% of the total tasks, as in the case of procedure CR708V07_VAR-INIT, and as low as 50% of the total tasks, as in the case of procedure 3000-WRITE-TOTAL-REPORT. Visual inspection of this code reveals the reason for this disparity. Procedure CR708V07_VAR-INIT is the main controller function which initialises many variables before calling initial procedures. Because this procedure carries out many initialisations of disparate and unrelated variables within the same procedure, a high degree of parallelism is possible. Furthermore, because the purpose of this function is to initialise variables and call starting procedures, there are no control constructs within this code.

Procedure 3000-WRITE-TOTAL-REPORT, on the other hand, contains a number of control constructs and shared variables among codelines. Consequently, the degree of parallelism is very much restricted.

The total degree of parallelism using codeline level of granularity, measured among all procedures, is about 90%.

4.4 Summary

The type of granularity greatly determines the type of number of tasks that are deemed to be able to execute independently. An example, at the procedural-level granularity, only 11% of the granular units (procedures) were deemed to be independent. At the programming block level, all of the programming blocks in 81% of the procedures were deemed able to execute independently. Similarly, at the code line level, 90% of the code lines were deemed to be able to execute independently.

At the program block level, there is an average of 4 parallel tasks per procedure. At the code line level, there is an average of 31 parallel tasks per procedure. At the procedural level, there are 4 parallel tasks. A procedure has an average of 34 code lines per procedure; a program block has an average number of code lines of 16.

If we divided the number of parallel tasks per evaluation unit (such as procedure, programming block, or individual code line), we obtain the following statistics:

1)Procedural-Level: 4 parallel tasks/37 procedures =0.11 parallel tasks per unit
2)CodeBlock: (995) 8 parallel tasks/64 program blocks =0.125 parallel tasks per unit
3)Individual Code Line: 1155 / 1279 code lines=0.90 parallel tasks per unit.

Given these statistics, it would seem that the finest level of granularity, at the code line level, offers the best level to determine independent tasks. However, at the same time, this code line level offers the widest degree of disparity, approximately 50%, between the degree of independent tasks found. The degree of parallelism found, at the code line level, seems to be highly dependent on the nature of the associated procedure's function (such as initialisation or writing to a file) as well as the number and degree of control constructs present.

5. Conclusions

In this study, we examined the degree of task independence among tasks of different granularities. The task granularities selected were procedural, programming blocks and individual lines of code. Because program design and programming style both affect the size of procedures and of programming blocks and determine the degree of interconnectedness between each granular unit.

Our investigation revealed that the individual codeline is the optimal granular unit of tasks for determining task independence. Although the average number of codelines per procedure is quite small, the program seems to have been designed such that the procedures operate in a rather sequential fashion, with a degree of interconnectedness between the procedures. Although some procedures have been identified as being able to execute in parallel, most of these procedures have been designed to operate sequentially. Similarly, program blocks are slightly less efficient than individual codelines in determining and optimising task independence. The reasons why individual codelines, as a granular unit, are optimal, in comparison with other granular units such as procedures and program blocks may lie with the design, and ultimately the nature, of this legacy

One of the main advantages that individual codeline granularity has over program block granularity is their ability to parallelise record field operations during record operations. COBOL programs typically involve numerous record operations and the records involved in these operations typically are large. Because all record fields of a record would be considered as part of a single program block according to program block granularity, the operation involving these record fields would be serialised and not be parallelised in any way. However, according to individual codeline granularity, the operation involving each record field of a record could be parallelised. Because individual fields of a record operation typically share no control or data dependencies among themselves, all the I/O operations involving the fields of the same record could be executed in parallel.

6. References

[1] Ammarguellat, Z., "A Control-Flow Normalization Algorithm and Its Complexity", IEEE Transations on Software Engineering, Vol 18, No 3, March 1992, pp. 237 – 251.

[2] Bisbal, J., D. Lawless, B. Wu and J. Grimson, "Legacy Information Systems: Issues and Directions", IEE Software, Sept/Oct 1999, pp 103-111.

[3] Dietrich, W., L. R. Nackman and F. Gracer, "Saving a Legacy With Objects", Proceedings of OOPSLA '89, ACM, 1989, pp 77-83.

[4] Eisenbarth, T., R. Koschke and D. Simon, "Aiding Program Comprehension", Technical report, Universitat Stuttgart, Stuttgart, Germany.

[5] Gall, H., R. Klosch and R. Mittermeir, "Architectural Transformation of Legacy Systems", 17th International Conference on Software Engineering (ICSE-17) Workshop on Program Transformation for Software Evolution, Technical Report Number CS95-418, Seattle, USA, 1995.

[6] Hausler, P., A, Mark, G. Pleszkoch, R. C Linger and A. R. Hevner "Function Abstraction to Understand Program Behavior", IEE Software, Jan 1990, pp 55-63.

[7] Lam, M., S. Robert and P. Wilson, "Limits of Control Flow on Parallelism", Proceedings of the 19th Annual International Symposium on Computer Architecture, Gold Coast, Australia, 1992, pp 46-57, ACM.

[8] Liu, X., "Abstraction: A Notation for Reverse Engineering", Ph.D. Thesis, De Montfort University, England, 1999.

[9] Millham, R., "An Investigation: Reengineering Sequential Procedure-Driven Software into Object-Oriented Event-Driven Software Through UML Diagrams", Proceedings of the 26th Annual International Computer Software and Applications Conference (COMPSAC) 2002, IEEE Computer Press, Oxford, England.

[10] Muller, P., " Instant UML", Wrox Press, Birmingham, UK, 1997, p 131.

[11] Rich, C. and L. M. Wills, "Recognizing a Program's Design: A Graph-Parsing Approach", IEE Software, Jan 1990, pp. 82 – 89.

[12] Software Reuse Technnology Roadmap v 2.2, Volume 1 Available at http://sw-eng.falls-church.va.us/ReuseIC/pol-hist/Roadmap/Vol1/V1-3.html

[13] Tsai, W.T. and J.K. Joiner, "Re-engineering Legacy COBOL Programs", www.acm.org/cacm/extension/joinertx.pdf

[14] Ward, M., "The Syntax and Semantics of the Wide Spectrum Language", *Technical Report*, Durham University, England, 1992.

[15] Sneed, H., "Migration of Procedurally Oriented COBOL Programs in An Object-Oriented Architecture", International Conference Software Maintenance, IEEE Press, Orlando, 1992.

[16] Levey R., *Reengineering Cobol With Objects : Step by Step to Sustainable Legacy Systems*, McGraw Hill, January 1996.

[17] van Deursen, A. and T. Kuipers, "Identifying Objects using Cluster and Concept Analysis", International Conference on Software Engineering, IEEE Press, 1999.

[18] Yang, H., *Acquiring Data Designs from Existing Data-Intensive Programs*, PhD Thesis, Durham University, England, 1994.

[19] Ward, Martin "Specifications from Source Code -- Alchemists' Dream or Practical Reality?" 4th Reengineering Forum, September 19-21, 1994, Victoria, Canada

[20] Chandy, K. Mani, Carl Kesselman The Derivation of Compositional Programs (1992) Joint International Conference and Symposium on Logic Programming. Available at http://citeseer.nj.nec.com/chandy92derivation.html

[21] Joiner, J K, W T Tsai "Reengineering Legacy Cobol Programs". Available at www.acm.org/cacm/extension/joinertx.pdf

[22] Z. Ammarguellat. "A control-flow normalization algorithm and its complexity". IEEE Transactions on Software Engineering, 18(3):237--251, March 1992.