SOFTWARE MAINTENANCE, REVERSE ENGINEERING AND REENGINEERING

Many sources agree that programmer efforts are mostly devoted to maintaining systems (1). Pressman (2) estimates that a typical software development organization spends anywhere from 40% to 70% of all dollars conducting maintenance. (This is not surprising when one considers the quantity of code that must be maintained in these legacy systems; it was estimated in 1990 (3) that there were 120 billion lines of source code in existence.) A large portion of the maintenance effort is spent understanding the code under maintenance. Previous studies have shown that more than 50% of perfective and corrective maintenance effort is spent trying to understand existing programs. This involves reading the documentation, scanning the source code, understanding the changes to be made, and so on (4).

However, most of the legacy systems were developed before software engineering techniques were widely used. In general, they are ill-structured and their documentation is poor, out-of-date, or totally absent. In part, this lack of documentation stems from the fact that software documentation is usually the last priority in the development effort. In addition, with the modification of code, the original documentation may or may not have been modified to keep it current with the code. Consequently, the original documentation, if it exists, may be inaccurate, incomplete, and inconsistent with regard to the code under maintenance.

Due to the lack of reliability of software documentation, the only documentation that software maintainers assume is reliable is the source code of the system they are supposed to maintain. However, the code may have been subjected to a large number of changes over the years (even decades) and, thus it presents a high level of entropy (i.e., ill-structured, highly redundant, poorly self-documented, and weakly modular). High levels of entropy combined with imprecise documentation make software maintenance difficult, time-consuming, and costly.

In order to improve maintenance, it is important to develop tools, techniques, and methods for assisting in the process of understanding existing software systems. Trying to understanding the program is the process that consumes most of the maintenance efforts. It consists of acquiring knowledge about a software system. Broadly speaking, the process of learning about a software system involves reverse engineering of the source code to identify the system's components and their interrelationships. Chikofsky and Cross (5) define reverse engineering as backward engineering of a system to the specification stage. It is then the opposite process of conventional engineering, where the system is synthesized from high-level specifications and conceptual, implementation-independent designs and then physically implemented. Figure 1 illustrates this concept. Generally, we consider reverse engineering to be the process of analyzing an existing system in order to identify the system's components and their relationships, and to create representation of the system in a more intelligible form or at a higher level of abstraction. The key idea here is to move from a concrete representation of the system to an abstract and intelligible one without changing the existing system. The aim is to discover high-level concepts (e.g., design strategies and business rules) from software artifacts and then to use those concepts to improve software maintenance. To do so, we can take advantage of other information in addition to the source code (e.g., domain knowledge, programming knowledge, and documentation).

Reverse engineering does not require changing the system at all; this is the goal of reengineering. Reengineering is the examination and the modification of an existing system to reconstitute it in a new form, followed by the implementation of the new form. The first phase of reengineering is some form of reverse engineering so as to abstract and understand the

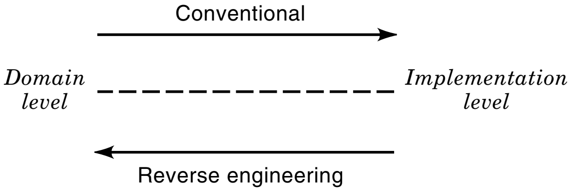


Figure 1. Reverse engineering versus conventional engineering.

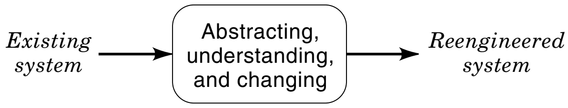


Figure 2. The reengineering process, which consists in extracting the relevant information from the source code, understanding it, and changing it to achieve the goals of the reengineering.

existing system. The second phase is traditional engineering or full restructuring using new specification and knowledge of the old system obtained from reverse engineering. This process, as illustrated in Fig. 2, is generally set into motion by the need to move old programs and systems to new platforms, as in source code translation, or restructuring programs that were corrupted by repeated maintaining activities. However, the most promising axis of reengineering is certainly moving legacy systems to emerging technologies and paradigms. Indeed, many organizations have been migrating their legacy systems to emerging technologies (e.g., object-oriented technology). Lehman and Belady (6) present this migration as an economical choice through their three laws on the evolution of large systems. The object-oriented (OO) paradigm is the target architecture of choice for the reorganization of systems, since OO representations are supposed to be much easier to understand than their classical structured counterparts. Furthermore, the encapsulation limits the complexity of maintenance. Any modification in the implementation of an object (class) is not supposed to generate side effects since only its interface is visible by the others objects.

OO approaches and languages have have become quite popular, partially because of their potential benefits in terms of maintenance (reusability, separation of concerns, and information hiding). However, the vast majority of software available today is not OO. The effort required to simply rewrite software from scratch using an OO approach would be prohibitive, and significant expertise recorded in the procedural software would be lost. The cost of manual conversion would also be prohibitive. A tool or a tool set that would support the conversion of procedural code to OO, even in a semiautomatic fashion, would ease the introduction of OO technology in many organizations. This kind of reengineering tool could be especially helpful to integrate existing systems and new ones developed with OO approaches.

Both reverse engineering and reengineering are related to the improvement of software development by producing solutions and resources to the maintenance of legacy systems. For both, the keyword remains "abstraction": defining a set of abstractions that allows us to represent the system under maintenance in different forms, depending on the targeted analysis subject. These abstractions will be exploited for generating documents of various types, for generating diagrams, and for giving information about data and control flow, which is the topic of reverse engineering. They also give us the chance to discover candidate objects in procedural code, reengineering it in an OO resulting system.

PROGRAM ABSTRACTION: THE KEYWORD FOR REVERSE ENGINEERING AND REENGINEERING

When a software maintainer maintains a program—in general, poorly documented—he or she follows a bottom-up process by detecting patterns indicating the intent of some portion of code. In order to comprehend the understanding process, it is important to look at human factors involved in this process. This area of research is called software psychology, and a variety of models of the human program understanding process have been developed: Shneiderman's model, Brooks' model, Soloway's model, and so on. References 7 to 10 include pertinent surveys on the topic.

Computers are much more rigorous and formal than humans. Thus, when we try to understand how a computer program could understand other programs, we talk about automated program understanding. By the use of automatic program analysis, we try to capture high-level concepts, such as software design diagrams, directly from the code. This analysis could be static or dynamic. Static analysis does not require the program to be executed. It involves examining the source code of programs or designs. Dynamic analysis evokes the process of systematically executing the programs in order to capture their performance and correctness properties. The most common forms of dynamic analysis are: profiling, testing, and partial evaluation. Profiling determines, for example, the number of times each statement or each procedure is executed. It works by adding an extra code to do so, or by periodically interrupting the executing program to determine what it is currently doing and then using a statistical model. Testing is the most common form of dynamic analysis. Of course, we need techniques for making sure that tests are realistic (11). For example, statement coverage ensures that every statement is executed, and branch or condition coverage measures the extent to which all branches or conditions are executed. Finally, partial evaluation is a technique that takes as input a program and values for certain of the program's input parameters. It produces as output a smaller program equivalent to the original on those parameters. It is of great interest to understand complicated real-time systems (12).

Another dimension through which pertinent works are distinguished is the level of involvement of domain experts in the maintenance process. Some techniques are called supervised, in the sense that, in addition to the source code, they need some knowledge about source languages, general programming techniques, and application domains to infer properties of software systems at several levels of abstraction. Studies discussed in Refs. 13 to 15 are examples of it. On the other hand, unsupervised methods have as input only the source code, although they need some domain knowledge to make some decisions.

Taking into account the cost and the availability of a domain expertise, it is often more efficient to choose an unsupervised approach. Often we do not have any choice: Expertise, documentation, and developers of the application under maintenance are not available at all! Figure 3 illustrates an unsupervised and static approach of reverse engineering and reengineering. It assumes that program abstraction is the key step for all reverse-engineering and reengineering efforts, including redocumentation, data and control flow analysis, and object identification in procedural code.

To extract abstractions from a source code, we need a language tool that processes the source code and produces some kind of output. The internal design of language tools, in most cases, is very similar: A parser obtains a string of tokens from the lexical analyzer and verifies that the string can be generated by the grammar. The results of a syntactic analysis are

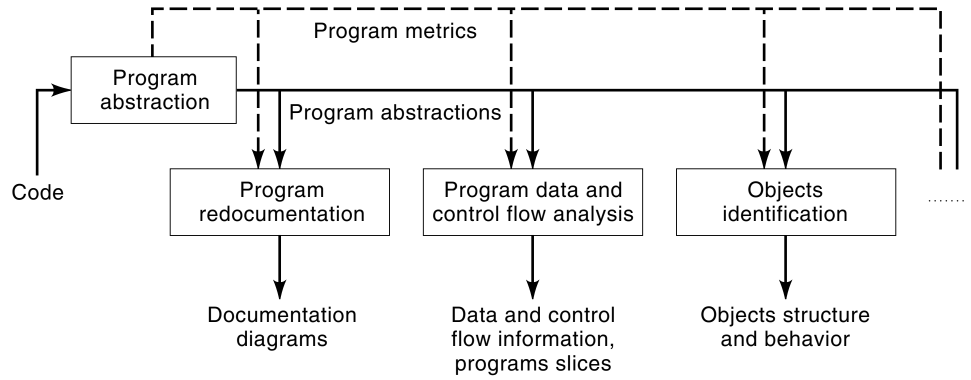


Figure 3. Overview of an unsupervised and static reverse-engineering and reengineering approach with three different goals: program redocumentation, program data and control flow analysis, and object identification.

represented by a tree structure. The more primitive version is called *parse tree*. It contains details not really related to program understanding, such as punctuation. An abstract representation of a parse tree leads to a structure called an *abstract syntax tree* (AST). The AST is the basis of more sophisticated program analysis approaches. Because an AST is a tree, its nodes can be visited in a certain sequence. This approach serves as the basis of many tools. They exploit the AST by performing operations on its elements. A common operation is pattern recognition that aims at finding in the AST all the occurrences of the patterns. Typical actions are then taken to generate the wanted abstractions. Figure 4 illustrates the common approach of such tools.

Abstractions for Reverse Engineering

In the following, we present the specifications of some abstractions of very low level. They are expressed in terms of predicates. By combining some of them, we can obtain abstractions of a higher level. Both program redocumentation techniques and program data and control analysis techniques can exploit them.

- *lmdm (l, x, other_attributes)*: This states that the datum *x* is of level *l*. This datum has attributes specified by "other_attributes" (e.g., in COBOL, REDEFINES, OCCURS, etc.).
- *lpdm (ln, pn, t, content)*: This states that the physical file "pn" is assigned to the logical name "ln" and that the file type is "*t*" and it contains "content."
- *struct (x, content)*: This states that the datum "*x*" is a structure and its fields are given by "content."
- *call (p, p_c_statement)*: This states that within the subprogram "*p*" there is CALL to another part of the program given by "*p*_c_statement."
- *s_dj (p, s, t)*: This states that within the subprogram "*p*" there is a statement group "*s*." All the statements of *s* are of type "*t*"—for example, *compute, call, input, output, condition,* and so on.
- *stmt_nd (p, s)*: This states that statement "*s*" is in subprogram "*p*."
- *pred_nd (p, s)*: This states that predicate (a condition in a conditional statement) "*s*" is in subprogram "*p*."
- *du (s1, s2, x)*: This states that datum "*x*" is defined in statement "*s*1" and used in statement "*s*2." A variable is defined when it is given a value in the statement. A statement uses it when its value is used in this statement.
- *def (p, s, x)*: This states that datum "*x*" is defined in statement "*s*" of subprogram "*p*."
- *use (p, s, x)*: This states that datum "*x*" is used in statement "*s*" of subprogram "*p*."
- *recdepth (x, d)*: This states that data structure "*x*" has depth "*d*."
- *cpdepth (d)*: This states that the call graph of the target program has depth "*d*."
- *nbrp (n)*: This gives the number "*n*" of subprograms in a program.
- *nbrbranch (p, n)*: This gives the number "*n*" of branching statements for each subprogram of the program.
- *nbrcompute (p, n)*: This gives the number "*n*" of computing statements for each subprogram of the program.
- *nbrctrl (p, n)*: This gives the number "*n*" of control points for each subprogram of the program.
- *nbrio (p, n)*: This gives the number "*n*" of input/output statements for each subprogram of the program.
- *varparg (x, ps)*: This gives for each variable "*x*" the list of subprograms "*ps*" where it appears.

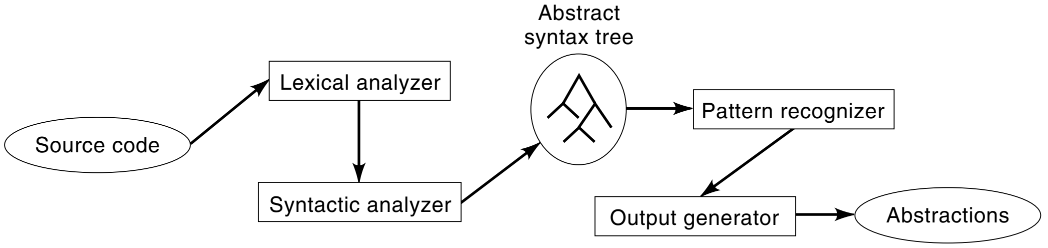


Figure 4. Generation of abstractions from the code through the abstract syntax tree.

- *varmanip (x, p, m)*: This gives for each occurrence of a variable "*x*" in a subprogram "*p*" the mode "*m*" of its manipulation. This mode can take the values *C*, *P*, *I*, *M*, or *T* (16,17).
  - *C*: *The data value is used in the right-hand side of an assignment or in an output statement.*
  - *P*: *The data value is used in the predicate part of a conditional statement.*
  - *I*: *The data value is first used to define an other data (C mode). This new datum is used in a P mode.*
  - *M*: *The data value is modified.*
  - *T*: *The data value is not modified, it is just passed through a CALL statement to another routine of the program.*

Some of the abstractions presented above are assigned to be used in a metric computation process. Such a process is used, for example, to obtain the profile of the application under maintenance, to predict the amount of effort needed in maintaining the applications, or to guide the generation of documentation diagrams. Table 1 illustrates this idea.

Table 1. Usefulness of Each Generated Abstraction

Abstractions	What Is Their Help for Program Redocumentation and Program Data and Control Flow Analysis?
lmdm (*l*, *x*, other_attributes)	Gives information about memory data and their relationships and helps generate a data model like diagram
lpdm (ln, pn, *t*, content)	Gives relevant files information of the software system and is exploitable for generating a file model like diagram
struct (*x*, content)	Describes data structure composition and helps generate a Warnier–Orr diagram
call (*p*, *p_c*_statement)	Helps generate a call graph
s_d*j* (*p*, *s*, *t*)	Gives a task-oriented summary of the software system and is exploitable to generate a Jackson diagram
stmt_nd (*p*, *s*)	Gives information about control and data flow in a def–use like graph
pred_nd (*p*, *s*) def (*p*, *s*, *x*), use (*p*, *s*, *x*)	Is exploitable by slicing algorithms in program data and control flow analysis
stmt_nd (*p*, *s*)	Gives a conceptually different information about control and data flow
pred_nd (*p*, *s*) du (*a*1, *s*2, *x*)	Is exploitable by slicing algorithms in program data and control flow analysis
varparg (*x*, *ps*) varmanip (*x*, *p*, *m*)	Is exploitable by a metrics-based slice validation process
recdepth (*x*, *d*), cpdepth (*d*), nbrparag (*n*), nbrcompute (*p*, *n*), nbrbranch (*p*, *n*), nbrio (*p*, *n*), nbrctrl (*p*, *n*)	Is exploitable by a metrics-guided redocumentation process

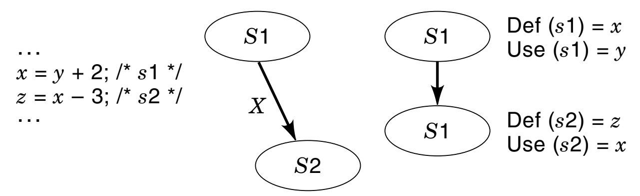


Figure 5. Higher-level abstractions: a program dependence graph and a def–use graph.

On the other hand, by combining some of them, we obtain abstractions of a higher level. The graphs presented in Fig. 5 are an example of such high-level abstractions.

Abstractions for Reengineering to Object Technology

The source code contains part of the knowledge about the application. To identify object-like features in it, we have to decide which information must be used. Depending on the existing techniques, there are different program abstractions that can be used for this purpose. For this section we limit ourselves to the following examples. The first two enable us to identify objects, while the third one enables us to identify classes.

1. *Routines Interdependence Graphs.* As proposed by Liu and Wilde (18), these graphs show the dependence between routines consequent to their common coupling to the same global data. A node $P(x)$ in the graph denotes the set of routines which reference a global variable x. An edge between $P(x_1)$ and $P(x_2)$ means that the two corresponding sets are not disjoint ($P(x_1) \cap P(x_2) \neq \emptyset$).

   Figure 6(a) shows the reference relationship between the routines *fi*s and the global data *di*s of a program. The *ti*s represent the global data types. Figure 6(b) gives the corresponding routines interdependence graph.
2. *Reference Graphs.* In such graphs, nodes are routines or global variables, and an edge between a routine and a variable means that the function uses the variable (19). Figure 7 shows the reference graph of the relationship of Fig. 6(a).
3. *Type Visibility Graphs.* As introduced in Ref. 20, such graphs represent the visibility relationship between the routines and the data structures (or types) of a program. A type t is said to be visible by a routine f if f uses a global variable of type t, if f has formal parameter of type t, or, finally, if f has a local variable of type t. Figure 8 gives a partial-type visibility graph based on the relationship in Fig. 6(a).

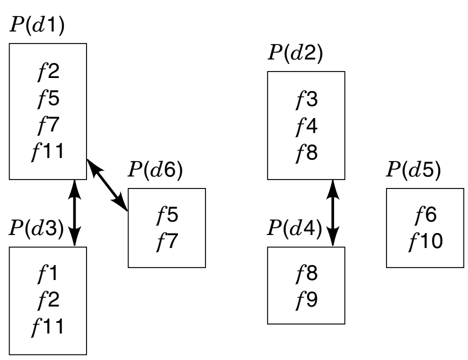
The next sections will show how the abstractions presented above are useful for reverse and reengineering legacy systems.

PROGRAM REDOCUMENTATION

When determining the true cost of a new software system, one important consideration is the estimation of the software system lifetime. Of great importance in this estimation is the

	$d1$ ($t1$)	$d2$ ($t1$)	$d3$ ($t2$)	$d4$ ($t3$)	$d5$ ($t3$)	$d6$ ($t4$)
$f1$			x			
$f2$	x		x			
$f3$		x				
$f4$		x				
$f5$	x					x
$f6$					x	
$f7$	x					x
$f8$		x		x		
$f9$				x		
$f10$					x	
$f11$	x		x			

(**a**)



(**b**)

Figure 6. (a) Reference relationship between routines and global data. (b) Routines interdependence graph of the relationship in part (a). Each node contains the set of routines that reference a given global data, and each edge indicates that the related sets overlap.

maintainability of the software system; it includes the adequacy of the programmer documentation. Software documentation is usually the last priority in the development effort. One reason for this is that developers try to get a product out the door before it is obsolete or before the market competition beats them. In order to extend the lifetime of a software system, some mechanisms must be found to make old systems easier to maintain or modify. One approach is to automatically generate documentation from the source code of the existing software system. Chen et al. (21) define redocumentation as follows:

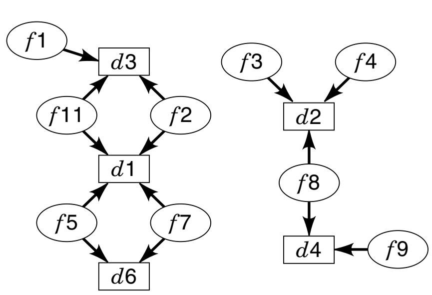


Figure 7. Reference graph of the relationship in Fig. 6(a). The routines are represented by ellipses and the data by rectangles.

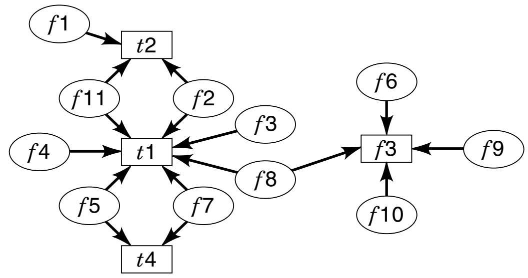


Figure 8. Type visibility graph of the relationship in Fig. 6(a). Each routine is related to the data types it can use.

Changing the related program documents including specification and design to reflect the program change.

Several approaches have been proposed in order to automatically generate software documentation that assists the understanding process and the recording of the results of this process. Some of these approaches generate informal documentation (15,22,23), and others generate formal and semantically sound documentation (24,25). In the following, we give a nonexhaustive list of documentation formalisms that are automatically feasible starting from the abstractions defined in the previous section. One approach is to translate the chosen abstractions to a graph description like language (e.g., Ref. 26) and then use a visualization tool to produce the diagrams.

Live Memory Data Diagram. This is a graph where nodes represent data and edges represent relationships between them. In COBOL programs, these relations are *contains, array-of, alias,* and *redefines,* and we say that an entity A is coupled with entity B through the relation $\mathscr{R}$, noted $A\ \mathscr{R}_r\ B$. Each relationship has the following meaning:

- *A Contains B*: This indicates that A contains B. In COBOL, for example, A would have a lower level number than B and it would be defined within A.
- *A Array-of B*: This indicates that entity A contains a table of entities of the type of entity B. This corresponds to the *OCCURS* keyword of COBOL.
- *A Alias B*: This indicates that the name A refers to the same entity as B. This corresponds to *LEVEL 88* in COBOL.
- *A Redefines B*: This indicates that entity A is an alias of B, but it redefines its structure. This corresponds to the keyword *REDEFINES* in COBOL.

Live Physical Data Diagram. This model is represented by a graph where nodes represent file data elements and properties, and edges represent relationships between them. The relationships between nodes of this graph are given by the abstraction *lpdm* (*lf*, *pf*, *t*, *r*). It states that physical file *pf* is assigned to logical name *lf*, and that the file type is *t* and it contains *r*. Figure 9 shows an example of such a diagram.

Warnier–Orr Diagram. This is a simple and straightforward technique for representing a software system structure and can be used either as a data-modeling tool or as a soft-

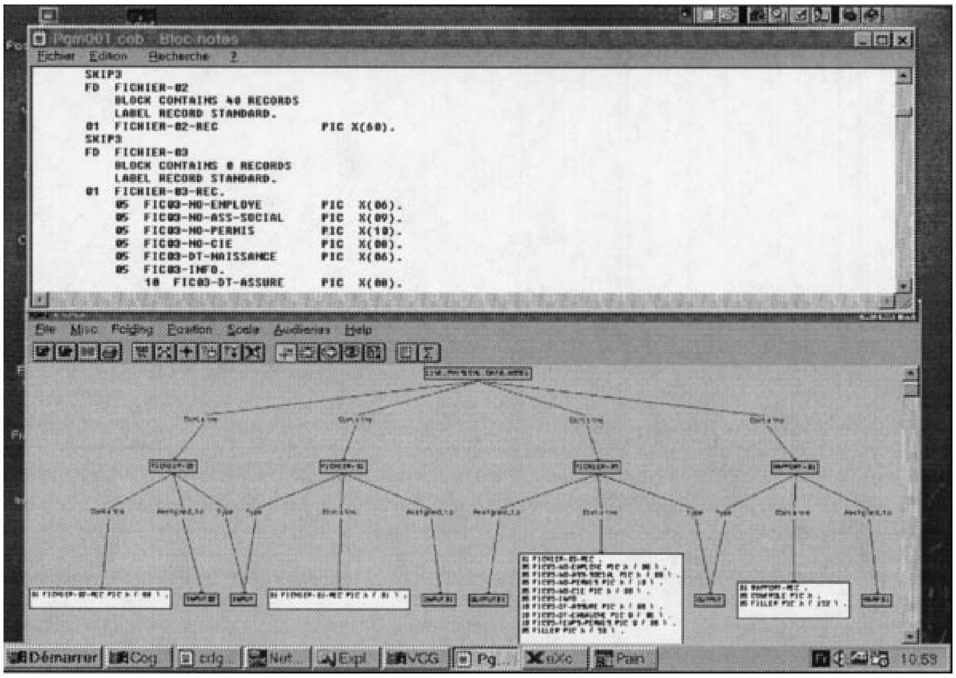


Figure 9. From the source code to a live physical data diagram, an example of automatic redocumentation.

ware module-structuring tool. It is most often used to describe data structure composition. The sequence of refinement is presumed to be left to right and top to bottom. This kind of diagram shows the composition of structures, calling hierarchies, data-structure definitions, or file format specifications. Figure 10 gives an illustration of this type of diagram.

Call Graph. In most understanding activities of programs and in particular in control analysis, it is helpful to know what the called subprograms are. In this topic, it is important to identify such information. It takes the form of CALL graph called also PERFORM-CALL graph for programs written in COBOL.

Jackson Diagram. This shows program operations, such as sequences and iteration among program modules. The entire program is represented as a hierarchical tree of boxes. The lower-level boxes show fine-grained sequences and iteration detail, and the higher-level boxes delineate program module organization. Another interesting topic concerns summarization of software systems. We call system summarization the techniques that factor out the portions of a system performing

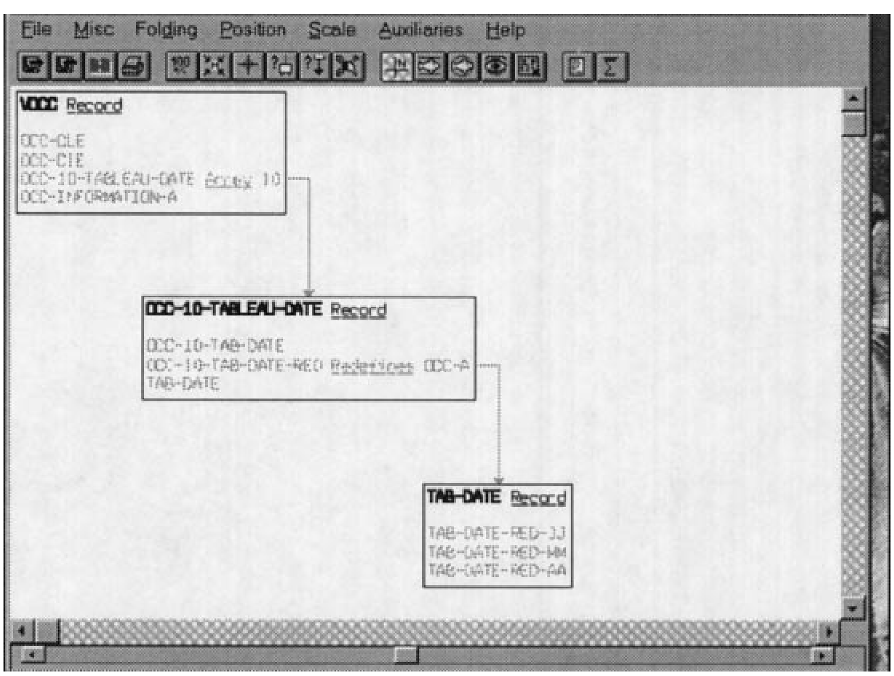


Figure 10. A Warnier–Orr diagram of a COBOL program. It identifies the data structures in a program.

certain tasks (e.g., database, interface, and communication) and present the relationships between them. For example, one could build a graph where a node represents an interface action, and the link between two nodes indicates that the action associated with one node can be executed before the action associated with the other node. This type of technique can be useful for understanding an aspect of a system, because it presents a summary of the system from that point of view.

Data Definition–Use-Oriented Graphs. The two last generated diagrams we are going to talk about are def–use graphs and program-dependence graphs. They are particular types of flow charts; both are very useful in data and control flow analysis, and they are inputs to most of slicing techniques. Thus the interest of these graphs is twofold: (1) They help the software maintainer in comprehending the software system and mainly the flow of variables, and (2) they give us the opportunity to carry out some slicing for extracting environment-dependent functions (such as operations on database or files, and report production) or domain-dependent functions (such as computational formulas and business rules).

A def–use graph is a quadruple $DUG = \langle G, V, D, U \rangle$, where $G = \langle N, E \rangle$ is the control flow graph representing the program, V is the set of variables in the program, and D and U are functions mapping N (the nodes of G) in the set of variables which are defined or used in the statements corresponding to nodes. Abstractions *stmt_nd (p, s)*, *pred_nd (p, s)*, *def (p, s, x)*, and *use (p, s, x)* of the section entitled "Abstractions for Reverse Engineering" help produce this type of graph. A program-dependence graph is a pair $PDG = \langle N, E \rangle$, where N is the set of nodes and E is the set of edges. The nodes are of three kinds: statement nodes, predicate nodes, and region nodes. There are two types of edges: control dependence and data flow edges. Abstractions *stmt_nd (p, s)*, *pred_nd (p, s)*, and *du(s1, s2, x)* are the basis for generating this type of graph.

Table 2 summarizes the help that the diagrams presented above can give to a software maintainer.

Table 2. Generated Diagrams and Their Help for Maintenance

Diagrams	What Is Their Help for Maintenance?
Life memory data diagram	Displays memory data and their relationships
Live physical data diagram	Displays relevant files information of the software system
Warnier–Orr diagram	Describes data structure composition
Call graph	Gives information about routines and their organization in a program
Jackson diagram	Gives a task-oriented summary of the software system
Def–use graph	Gives information about control and data flow Is exploitable by slicing algorithms
Program-dependence graph	Gives a conceptually different information about control and data flow Is exploitable by slicing algorithms

PROGRAM DATA AND CONTROL FLOW ANALYSIS

Background

Software maintainers are often constrained to study control and data flow in the software they are maintaining. Such a study is done thanks to techniques and algorithms developed for the data and control flow analysis area. There are two forms of *control flow analysis: intraprocedural* and *interprocedural.* The former determines the order in which statements can be executed within a subprogram. The latter determines the calling relationships among program units. Intraprocedural analysis aims at constructing a control flow graph (CFG). A CFG contains various symbols to represent different types of statements: assignments, procedure calls, conditions, and so on. The notion of basic block intervenes to construct a CFG; it is a maximal collection of consecutive statements such that the control can flow in only at the top and leave only at the bottom. Thus it corresponds to a node of the CFG. The utility of a CFG is that it gives an abstract picture of the ways in which a subprogram could run without entering the details of the statements of each basic block. The goal pursued by interprocedural control flow analysis is reporting invocations between subprograms belonging to a same software system. It often generates a call-graph, where the main routine is at the top. A node N is connected to a node M if routine represented by N calls routine represented by M. Arcs in the call graph are oriented.

There are many questions that control flow analysis cannot answer, such as those statements that may be affected by the execution of a given assignment statement. *Data flow analysis* is concerned with answering such questions. It is a more complex task than control flow analysis because it aims at answering questions related to how definitions flow to uses in a program and collecting information about potential executions of a program without actually executing the program. A usual way in which a variable is defined is when it occurs on the left-hand side of an assignment statement or when it occurs in a read statement. A use of a variable occurs when it is referenced, for example, in an arithmetic expression. For example, data flow analysis can discover if a variable remains a constant after an instruction of a program, determine which are the last statements in the program to assign a value to a particular variable before an instruction, or determine which values a variable can assume.

Most data-flow analysis comes from the area of compiler optimization. However, there is growing interest in using them in program understanding and maintenance. Data-flow information can be collected by setting up and solving systems of equations that relate information at different points in a program. A typical equation has the following form:

$$Out(S) = Gen(S) \cup (In(S) - Kill(S))$$

which signifies that information generated at the end of a statement S is generated within the statement or, alternatively, enters at the beginning and is not killed as control flows through the statement. If the control paths are evident from the syntax, then data flow equations can be set up and solved in a syntax-directed manner. An iterative method for computing reaching definitions works for arbitrary flow graphs, and its description is given in Ref. 27. Reaching definitions of a variable are often stored as *use–definition chains* or *ud-chains,* which are lists, for each use of the variable, of all the definitions that reach that use. Another kind of chain is the one called *definition–use chain* or *du-chain;* it contains the set of uses of a given variable, from a certain point in the program to another, so that there is no redefinition of the variable through this path.

Slicing: A Derivative Approach

Slicing is a derivative of data and control flow analysis. It is a family of techniques that indicate that a set of program statements are relevant. A statement $S1$ is relevant to a second statement S2 if it affects it directly or indirectly. A direct effect of $S1$ on $S2$ occurs when $S1$ defines a variable (i.e., assigns it a value) which is used in $S2$ or if $S1$ is a condition on the execution of $S2$. An indirect effect occurs when $S1$ affects directly or indirectly another statement $S3$ that affects directly $S2$.

The concept of *basic slice* has been introduced by Weiser (28). A slicing criterion is a tuple $C = \langle s, V\rangle$, where s is a statement and V is a set of variables. A slice with respect to C is a set of statements which may affect directly or indirectly the value of variables in V just before statement s. Another type of slice is defined by Ref. 29. It is called *direct slice* and it represents a subset of a basic slice. It considers only the statements that affect directly the value of variables in V before the execution of statement s. This kind of slice is used in identifying and extracting environment-dependent functions such as operations on database or files, report production, and so on (30), defined another type of slice called *decomposition slice.* It corresponds to the set of all the statements that contribute to the value of a variable at all the points in a program where the variable becomes visible outside the program. An example of such a point is a statement where the variable is displayed on screen or written in a file. It is built as the union of all the basic slices on the variable v with output statements of v and the last program statement, as statements in the slicing criterion. This kind of slice is used to extract domain-dependent functions, such as computational formulas or business rules. Lanubile and Visaggio (29) defined a *transformation slice* as the set of all statements that contribute to transform the values of input variables into the values of a set of output variables. Starting with the slicing criterion $\langle s, V_{\text{in}}, V_{\text{out}}\rangle$, it produces the set of statements that may affect directly or indirectly the value of variables in V_{out} before the execution of statement s starting with the value in V_{in}. Finally, Canfora et al. (31) defined the concept of *conditioned slice* corresponding to the set of all the statements that contribute to the value of a variable for a certain statement s, when a certain condition C holds. Its slicing criterion is given by $\langle s, V, C\rangle$.

The computation of a basic slice is based on a recursive definition. Let $C = \langle s, V\rangle$ be a slicing criterion and let G be the DUG associated to the program to analyze. $SUCC(n)$ is the set of successors of node n, $INFL(n)$ is the set of statements depending on a conditional statement n, $U(n)$ is the set of variables used in node (i.e., statement) n, and $D(n)$ is the set of variables defined in node n. The approach is recursive on the set of variables and statements which have either direct or indirect influence on V. Starting from zero, the superscripts represent the level of recursion.

- Step 0. The set of variables relevant to C, when program execution is at statement n, denoted $R_c^0(n)$, is defined as follows:

$$R_c^0(n) = \{v \in V / n = s\} \cup \{U(n)/D(n) \cap R_c^0(SUCC(n)) \neq \varnothing\} \\ \cup \{R_c^0(SUCC(n)) - D(n)\}$$

The set of statements relevant to C, denoted S_c^0, is defined as follows:

$$S_c^0 = \{n \in G/D(n) \cap R_c^0(SUCC(n)) \neq \varnothing\}$$

The set of conditional statements which control the execution of the statements in S_c^0, denoted B_c^0, is defined as follows: $B_c^0 = \{b \in G/INFL(b) \cap S_c^0 \neq \emptyset\}$

- Step $i + 1$

$$R_c^{i+1}(n) = R_c^i(n) \bigcup_{b \in B_c^i} R^0_{\langle b, U(b)\rangle}(n)$$
$$S_c^{i+1} = \{n \in G/D(n) \cap R_c^{i+1}(SUCC(n)) \neq \varnothing\} \cup B_c^i$$
$$B_c^{i+1} = \{b \in G/INFL(b) \cap S_c^{i+1} \neq \varnothing\}$$

The iteration continues until no new variables are relevant and so no new statements may be included. In other words, $S_c = S_c^{f+1}$, where f is an iteration step such that $\forall n \in N$, $R_c^{f+1}(n) = R_c^f(n) = R_c(n)$. Figure 11 gives an example of a basic slice.

Around a basic slicing algorithm, it is very helpful to define and implement other techniques related to data and control flow analysis in order to be able to perform different variants of slicing and of data and control flow analysis. Thus, considering the COBOL language, we are often confronted with programs written as a succession of paragraphs that represents a functional decomposition, and it is important to isolate all statements that are reached starting from a PERFORM statement. In this case, the criterion is only the PERFORM statement $\langle s\rangle$. Figure 12 gives an example of such a slice. To localize some functions implemented by the program and ultimately to transform the extracted functions to reusable ones, we can consider the transformation slicing: Starting with a slicing criterion $\langle s, V_{\text{in}}, V_{\text{out}}\rangle$, the goal is to isolate a transformation slice as the set of all the statements that contribute to transform the values of input variables V_{in} into the values of a set of output variables V_{out}.

When we isolate slices such as those presented in Figs. 11 and 12, a subsequent step is to extract and transform them to reusable components. Such components could be evaluated thanks to a metric computation process based on some abstractions produced in the section entitled "Abstractions for Reverse Engineering." This process would measure, for instance, a software quality attribute: cohesiveness.

REENGINEERING TO OBJECT TECHNOLOGY

As stated by Jacobson and Lindstrom (32), the process of reengineering can be defined by the simple formula

$$\text{Reengineering} = \text{Reverse engineering} + \text{Changes} \\ + \text{Forward engineering}$$

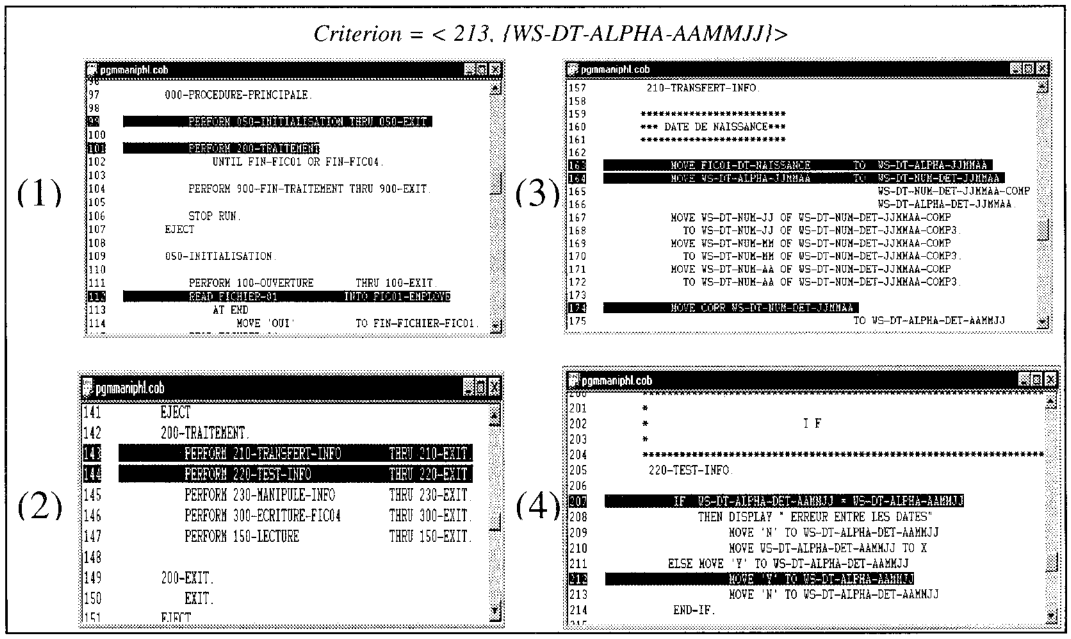


Figure 11. Obtaining a basic slice starting from the code. The highlighted statements are those that are affected by the slicing criterion.

"Reverse engineering" is the activity of defining a more abstract and easier to understand representation of the system. "Changes" have two major dimensions, namely, change of functionality and change of implementation technique. "Forward engineering" is the activity of creating a representation that is executable.

In the particular case of migration to object paradigm, the three elements of reengineering are more specific, and the formula above can be written as follows:

$$\text{Reengineering} = \text{Program abstraction} + \text{Object identification} + \text{Code generation}$$

In this section, we present some work on the migration of legacy systems to object paradigm. Globally, there are three families of approaches. The first family combines the normal engineering process (analysis, design, and implementation) with a reverse-engineering process. The two last families rely uniquely on the code.

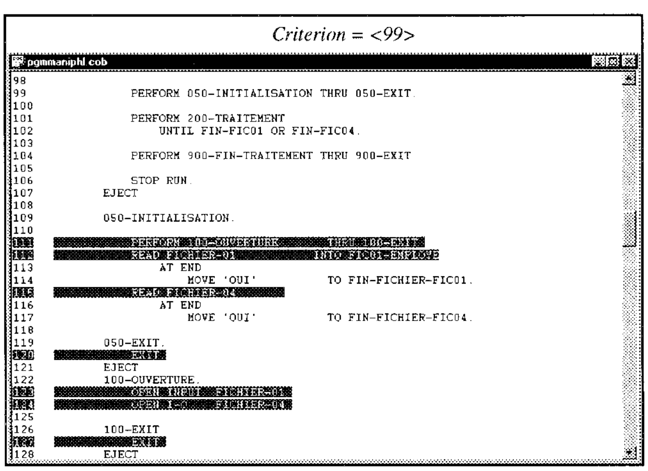


Figure 12. A PERFORM slice of a COBOL program.

Domain Knowledge Approaches

Based on the hypothesis that source code does not contain enough information to identify objects, this family of methods uses additional domain knowledge. To illustrate this trend, we present the COREM project (33). In this project, the migration to object technology is seen as a four-step process (see Fig. 13).

The first step is *design recovery*. In this step, different low-level design documents (i.e., structure charts, data-flow diagrams) are extracted from the source code. These documents lead to the generation of an entity-relationship diagram (ERD). The ERD is transformed into an object-oriented application model [called "reversely generated object-oriented application model" (ROOAM)], based on the structural similarities of these two design representations: Each entity is mapped to a ROOAM-object, and the corresponding *is-a* or *part-of* relationships are taken as *gen/spec* and *whole/part* structures, respectively. Objects and their attributes are directly derived from the entities of the ERD. The tentative ROOAM consists of static aspects only: No services or service relationships (message connections) are include yet.

Application modeling is the second step of the migration process. Based on the requirements analysis of the procedural input program, an object-oriented application model [called forward generated object-oriented application model (FOOAM)]is generated. The object-oriented application modeling process is done by a human expert who is experienced in the application domain or who participated in the development of the program under consideration. This modeling can be done by applying different object-oriented analysis methods.

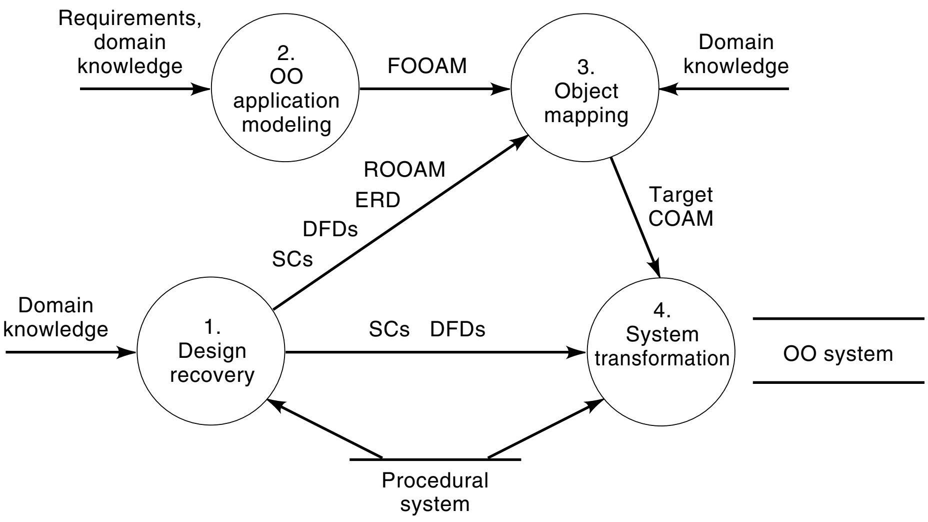


Figure 13. The COREM migration process. As we can notice, the domain knowledge is the key factor in the success of the process.

In the third step of the migration process (called "object mapping"), the elements of the ROOAM are mapped to the elements of the FOOAM, resulting in a target application model (target OOAM). The target OOAM represents the desired object-oriented architecture and is defined as the synthesis of the FOOAM and the ROOAM. It incorporates all elements that can be mapped between the two application models.

The final step (called source-code adaptation) completes the program transformation process on the source-code level and is based upon the results of the previous steps, especially the target OOAM.

A similar method was proposed by Shin (34). The main difference with COREM is that it uses the reference graph (see the section entitled "Abstractions for Reengineering to Object Technology") to construct the ROOAM.

Graph-Based Approaches

Liu and Wilde (18) have proposed two algorithms: one to group the data structures with routines that use them as parameters or return values, and the other to group the global variables with routines. The latter uses the routines interdependence graph (see the section entitled "Abstractions for Reengineering to Object Technology"). Each strongly connected subgraph is identified as an object. Later works (35–37) proposed some heuristics to enhance Liu and Wilde's work. Yeh et al. (20) combined data structures with global variables in order to form groups of routines, data structures, and global variables. Other algorithms use reference graphs as introduced in Ref. 19 (see section entitled "Abstractions for Reengineering).

One algorithm that illustrates this family is proposed by Canfora et al. (38). It decomposes a reference graph into a set of strongly connected subgraphs. In an object-oriented program, each object can be represented in the reference graph by an isolated subgraph. In a procedural program, this is not generally true because routines access data of more than one object. The goal of this algorithm is to decompose a reference graph into a set of isolated subgraphs by detecting undesired edges.

The algorithm in its original version works on a reference graph. Let F be the set of routines, let D be the set of data (depending on the used abstraction), and let E be the set of edges directed from routines to data. The PreSet of a data node is the subset of routine nodes that have an edge with this node. In the same way, The PostSet of a routine node is the subset of data nodes that have an edge with this node. Each $f \in F$ defines a subgraph that contains all the data nodes referenced by f and all the routines that only access these nodes. The subgraph of a routine is characterized by a measure of the internal connectivity called IC(f). The index IC(f) of a routine f is the ratio between the number of incident and internal edges of the subgraph of f. Formally,

$$\mathrm{IC}(f) = \frac{\sum_{d \in \mathrm{PostSet}(f)} \#\{f_i | f_i \in \mathrm{PreSet}(d) \wedge \mathrm{PostSet}(f_i) \subset \mathrm{PostSet}(f)\}}{\sum_{d \in \mathrm{PostSet}(f)} \#\mathrm{PreSet}(d)}$$

The IC(f) of a routine f allows us to compute the ΔIC(f). This value denotes the variation of the internal connectivity consequent to the clustering of the subgraph of f. Formally, ΔIC(f) is defined as follows:

$$\Delta\mathrm{IC}(f) = \mathrm{IC}(f) - \sum_{d \in \mathrm{PostSet}(f)} \frac{\#\{f_i | \mathrm{PostSet}(f_i) = \{d\}\}}{\#\mathrm{PreSet}(d)}$$

The decomposition of a graph into a set of isolated subgraphs is done through a series of steps. For each step, a step value SV is computed. SV is the threshold value for a ΔIC of a routine f that determines how to act upon the subgraph of f. Two actions are possible. The first one, *Merge,* means that all the data of the subgraph is clustered in a single data node. This action is done when ΔIC(f) ≥ SV (case of a routine that implement a method of an object). The second action, *Slice,* consists of slicing the routine f to dissociate two subgraphs. This occurs when ΔIC(f) ≤ SV (case of a routine that links together two objects). After each step a new set of routines (and a set of ΔICs) is obtained and a new step value is computed.

To illustrate this algorithm we use an example introduced in Ref. 38 (call it *collections*). This example presents a pro-

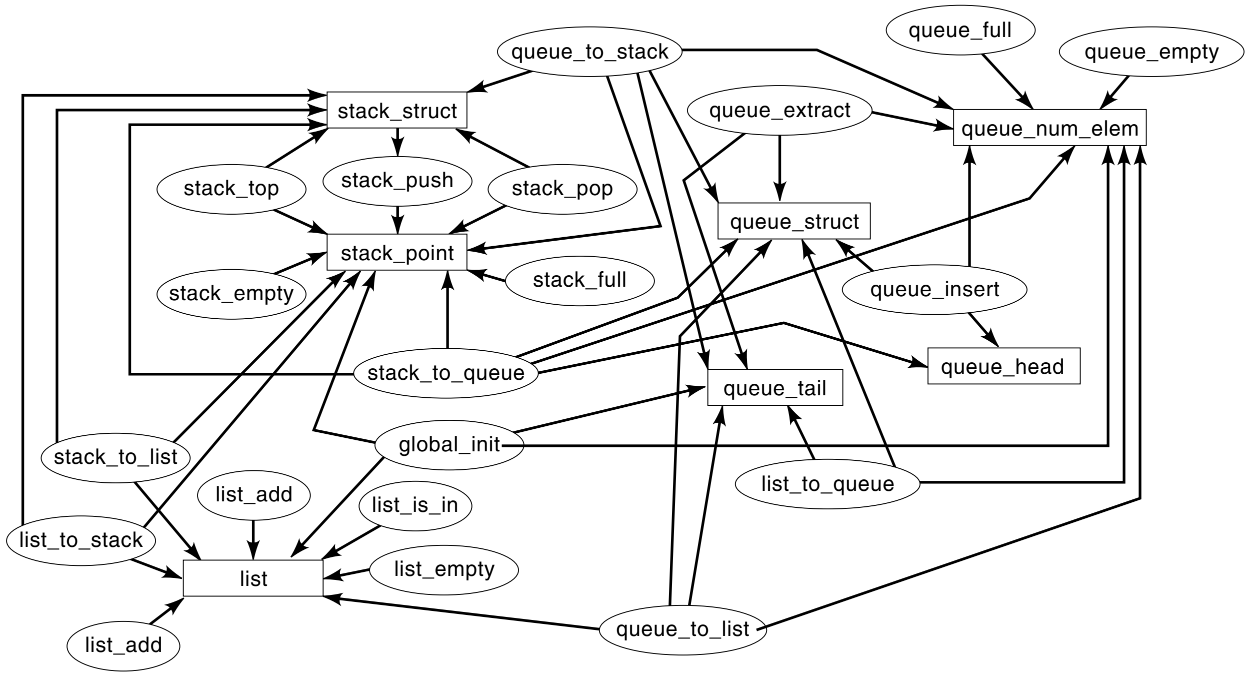


Figure 14. Reference graph for *collections* program. This program manipulates a stack, a queue, and a list. This graph is used as input in the processes of identifying objects.

gram that manipulates a stack, a queue and a list. Figure 14 shows the reference graph extracted from this program.

Using a statistical approach to compute the step value, the obtained value is SV = 0.14731. The set Merge is {*stack_push, stack_pop, stack_top*} and the set Slice is {*queue_insert, queue_extract, stack_to_list, stack_to_queue, queue_to_stack, queue_to_list, list_to_stack, list_to_queue, global_init*}. Variables to merge are *stack_struct* and *stack_point*. Routines really sliced are {*stack_to_list, stack_to_queue, queue_to_stack, queue_to_list, list_to_stack, list_to_queue, global_init*}.

The obtained graph is given in Fig. 15.

The second iteration gives the following results: SV is equal to 0.082992. The set Merge is {*queue_insert, queue_extract, stack_to_queue_B, queue_to_stack_B*}. The set Slice is {*queue_to_list, list_to_queue, global_init_B*}. Variables to merge are *queue_struct, queue_head, queue_tail,* and *queue_num_elem*. Routines really sliced are {*queue_to_list, list_to_queue, global_init_B*}.

The obtained graph is given in Fig. 16. It represents the final state of the reference graph. There are three isolated subgraphs corresponding each to an object (i.e., stack, queue, and list).

Concept Formation Approaches

Concept formation methods have been applied in software engineering for remodularization (see Refs. 39 and 40). In these two works, Galois (concept) lattices are used to identify modules in legacy code. The modules can be seen as objects in the sense that a set routines forms a module if they share the same data. The same technique is used to identify object (41). In the remainder of this section, we present this approach. This approach relies heavily on the automatic concept formation (42). It is based exclusively on information extracted directly from code.

Principle of Galois Lattice. We start by presenting the basic definitions for Galois lattices, proposed by Godin et al. (42). A better coverage of this subject can be found in Ref. 43. Algorithms based on this method are described in Ref. 44.

Let us take two finite sets E and E' and a binary relationship R between the two sets. The Galois lattice (see example in Fig. 17) is the set of elements (X, X'), where $X \in P(E)$ and $X' \in P(E')$. $P(S)$ is the powerset of S. Each element (X, X') must be complete.

A couple (X, X') from $P(E) \times P(E')$ is complete if it satisfies the two properties:

1. $X' = f(X)$, where $f(X) = \{x' \in E' | \forall\, x \in X, xRx'\}$
2. $X = f'(X')$, where $f'(X') = \{x \in E | \forall\, x' \in X', xRx'\}$

Given two elements $N_1 = (X_1, X_1')$ and $N_2 = (X_2, X_2')$ of a Galois lattice G, $N_1 < N_2$ implies that $X_2 \subset X_1$ and $X_1' \subset X_2'$.

This property defines a partial order between elements of G. A graph is constructed using this partial order [see Fig. 17(b)]. There is an edge between N_1 and N_2 if (1) $N_1 < N_2$ and (2) there does not exist $N_3 | N_1 < N_3 < N_2$. N_1 is said more general than N_2. Edges are directed from up to down.

Applicability to Object Identification. In an object-oriented design, an application is modeled by a set of objects where objects are composed of a set of data and a set of operations that manipulate these data. Most graph-based approaches to object identification group data with the routines that use them. Using this grouping approach, Galois lattices can provide all significant groups. Let E (see section entitled "Principle of Galois Lattice") be the set of global variables, let E' be the set of routines, and let R be the relation defined as $\forall\, v \in E$ and $\forall\, f \in E'$; if vRf means that the function f uses (refers

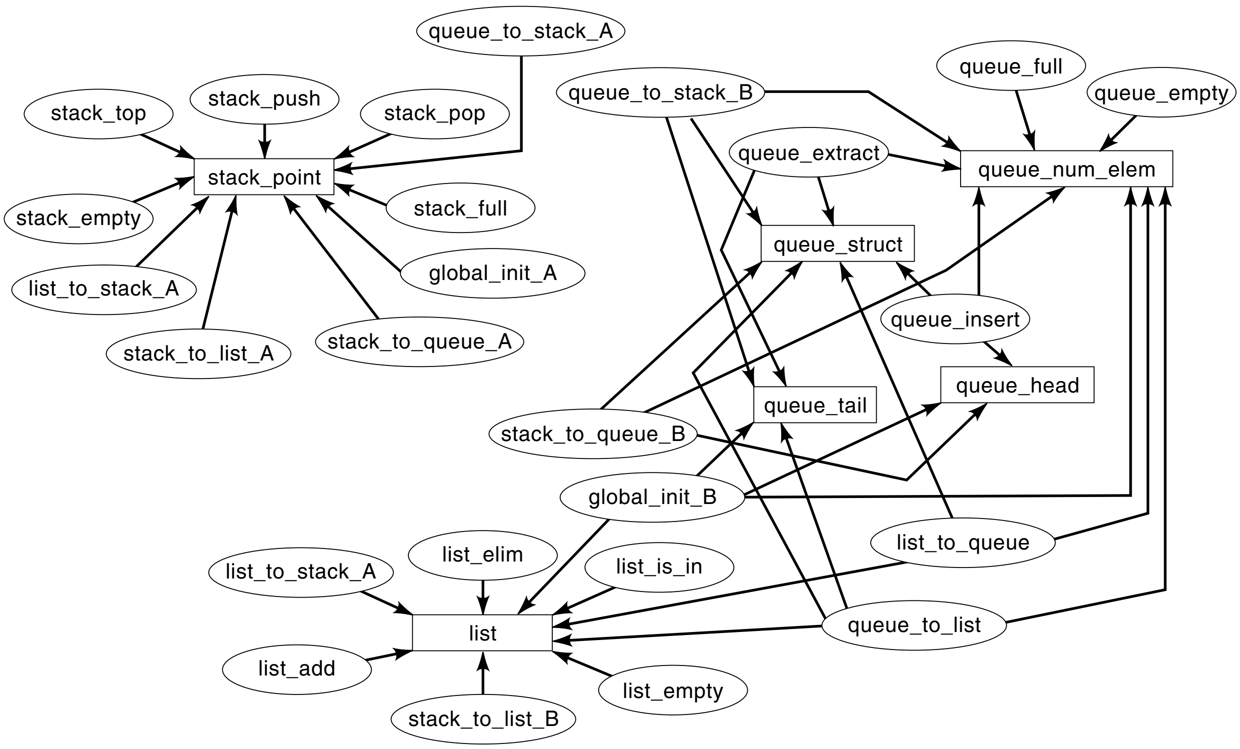


Figure 15. The reference graph after one iteration. An isolated subgraph appears. It represents the object stack.

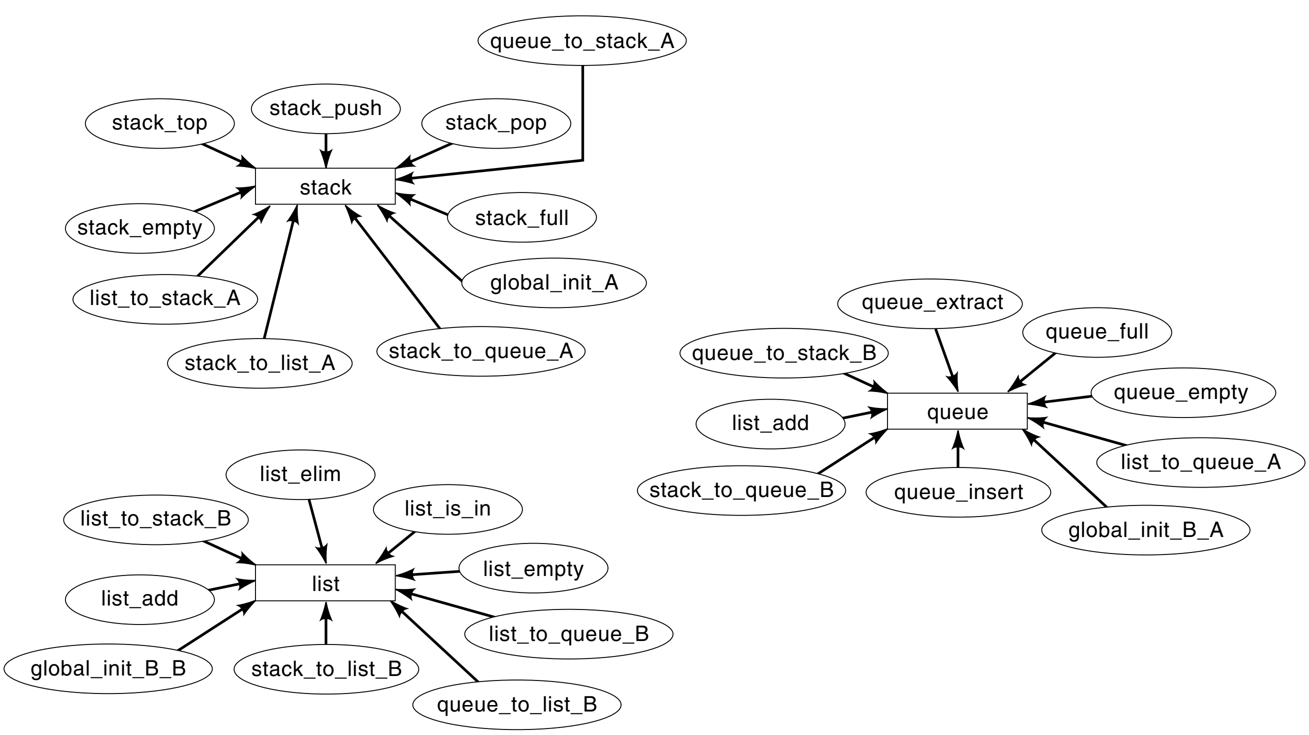
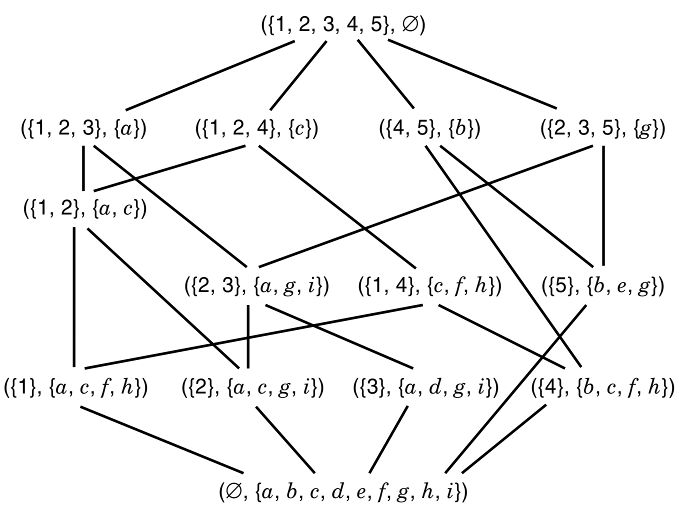


Figure 16. The reference graph after two iterations—final state. Each isolated subgraph represents an object.

		E'								
	R	***a***	***b***	***c***	***d***	***e***	***f***	***g***	***h***	***i***
	1	1	0	1	0	0	1	0	1	0
	2	1	0	1	0	0	0	1	0	1
E	**3**	1	0	0	1	0	0	1	0	1
	4	0	1	1	0	0	1	0	1	0
	5	0	1	0	0	1	0	1	0	0

(a)



(b)

Figure 17. (a) Representation of binary relation R. (b) Galois lattice for relation R.

to) the variable v, then the resulting Galois lattice has the following properties:

1. Each node (X, X') denotes a group of data (X) relative to a set of functions (X') which can be taken as a candidate object (the criteria are defined in the section entitled "Algorithm Steps").
2. There does not exist $(Y, Y') \neq (X, X') | Y \subseteq X$ and $Y' \subseteq X'$. Only significant groups are in the lattice.
3. An edge between two nodes $N_1 = (X_1, X_1')$ and $N_2 = (X_2, X_2')$ can be interpreted as either (a) a generalization/specialization link [from a behavioral point of view, the set of functions in N_1 is a subset of the set of functions in $N_2(X_1' \subset X_2')$] or (b) an aggregation link [from a data point of view, the set of data in N_2 is a subset of the set of data in $N_1(X_2 \subset X_1)$].

Algorithm Steps

Candidate Object Identification. As presented above, E is the set of global variables, E' is the set of functions, and R is the relation which indicates that $v \in E$ is used by $f \in E'$. Figure 18 shows the matrix representation of R' instead of R (for the *collections* program) for readability reasons. For the same reasons, names of functions and global variables are replaced by codes (number for a function and letter for variables) when building the Galois lattice.

The Galois lattice constructed from R presents all the significant groups of data (see Fig. 19 for the *collections* program). The goal of this step is to identify candidate objects. To this end, we define some criteria to select a subset of groups.

In order to identify candidate objects from the Galois lattice, we first define the set NS that contains the not-yet-selected variables. In the initial state, $NS = E$. The identification process stops when $NS = \emptyset$. In the identification process, groups are checked starting from the bottom up. This order is motivated by the fact that the deeper a group is in the lattice, the higher the cardinality of its function set (X'). In other words, our hypothesis is that a group of variables can be considered as a candidate object if these variables are simultaneously accessed by a large number of functions. In case of a tie (same cardinality of functions sets), groups are ordered by the cardinality of their variables sets (X) in a descendant mode. This is done to avoid large objects. These two criteria define a static order. If two groups have the same rank in this order, a priority is given to the one that has the higher cardinality of the set $ns = X \cap NS$. This defines a dynamic order. Each time a group is selected, the variables it contains are removed from NS. A group with $ns = \emptyset$ is ignored. The last criterion for selection is if a group has only one variable, the type of this variable must be nonbasic type (e.g., int and char).

The application of these criteria to the example of Fig. 19 gives the following four candidate objects:

$$co1 = \{b, c\} = \{stack\_struct, stack\_point\}$$
$$co2 = \{d\} = \{list\}$$
$$co3 = \{f, g, h\} = \{queue\_head, queue\_struct, queue\_num\_elem\}$$
$$co4 = \{e, g, h\} = \{queue\_tail, queue\_struct, queue\_num\_elem\}$$

Object Identification. If we consider candidate objects co3 and co4, we notice that they share two variables out of three. Such situations motivate the introduction of a new step that automatically merges these two objects. To detect these situations, we apply the same technique (Galois lattice) with a new relation. In this step, E is the set of candidate objects found in step 2. E' is the set of global variables. We define the relation R as $\forall\, g \in E$ and $\forall\, v \in E'$, gRv means that g contains v.

Figure 20 shows that co3 and co4 can be grouped in the same object. This decision is made relative to the cardinality of the set of variables in ({co3, co4}, $\{g, h\}$) which is fixed to 2 by default in our prototype. However, in our prototype an expert can be involved to make decisions based on his/her knowledge about the application domain, like merging candidate objects, or breaking a candidate object in two or more objects.

In the *collections* program example, we obtain the following objects:

$$o1 = co1 = \{b, c\} = \{stack\_struct, stack\_point\}$$
$$o2 = co2 = \{d\} = \{list\}$$
$$o3 = co3 \cup co4 = \{e, f, g, h\} = \{queue\_tail, queue\_head, queue\_struct, queue\_num\_elem\}$$

Method Identification. So far, we have identified the structure of the objects (variables). To be complete, an object must

R'	b. stack_struct	c. stack_point	d. list	e. queue_tail	f. queue_head	g. queue_struct	h. queue_num_elem
2. stack_push	1	1					
3. stack_top	1	1					
4. stack_pop	1	1					
5. stack_empty		1					
6. stack_full		1					
7. stack_to_queue	1	1			1	1	1
8. global_init		1	1	1	1		1
9. list_is_in			1				
10. list_empty			1				
11. stack_to_list	1	1	1				
12. list_to_stack	1	1	1				
13. list_add			1				
14. list_elim			1				
15. queue_to_stack	1	1		1		1	1
16. queue_extract				1		1	1
17. queue_full							1
18. queue_empty							1
19. queue_insert					1	1	1
20. list_to_queue			1		1	1	1
21. queue_to_list			1	1		1	1

Figure 18. Matrix representation of reference graph for *collections* program. The matrix is used as input to build the Galois lattice.

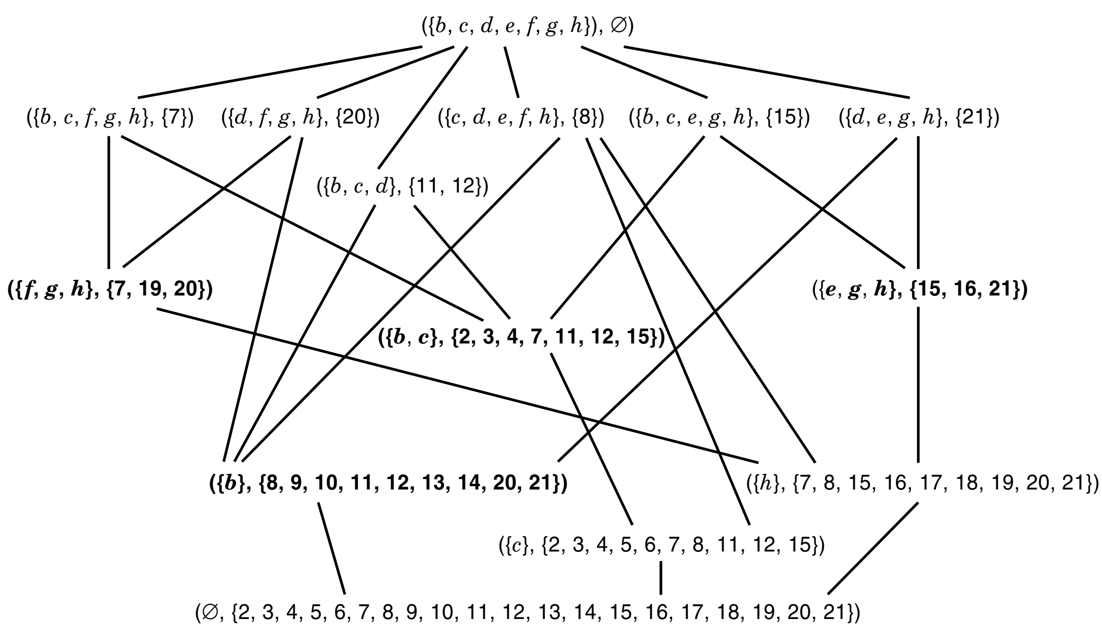


Figure 19. Galois lattice for reference relation (*collections* program). The nodes selected as candidate objects are represented in bold.

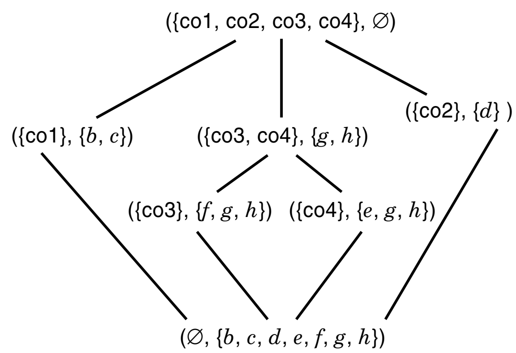


Figure 20. Galois lattice for grouping relation (*collections* program). The nodes co3 and co4 are grouped in a single object.

have a behavior (i.e., methods). In our approach, we identify methods from functions. In the remainder of this section, we present an overview of the rules we use to form methods from procedures/functions. A detailed description of method identification process is beyond the scope of this article. Some ideas we exploit can be found in Ref. 45.

Let O be the set of identified objects, let F be the set of functions in the legacy code, and let V be the set of global variables. For each function f, we define two sets $ref(f)$ and $modif(f)$ as follows: $\forall f \in F$,

$ref(f) = \{o_i \in O | \exists\ v_j \in V \textit{ and } v_j \textit{ in } o_i \textit{ and } v_i R f\}$, where R denotes the relation *is used by.*

$modif(f) = \{o_i \in O | \exists\ v_j \in V \textit{ and } v_j \textit{ in } o_i \textit{ and } v_i M f\}$, where M denotes the relation *is modified by.*

The relation M is derived from R with the condition that the mode of usage is modification.

There are three possible cases:

1. cardinality of ref(f) = 1
2. cardinality of ref(f) > 1 and cardinality of modif(f) = 1
3. cardinality of modif(f) > 1

For each case we define a rule.

Rule 1. For a function f, if cardinality of *ref(f)* = 1, then f becomes a method of the unique object in *ref(f)*.

The first case is trivial. For example, in *collections, ref(stack_full)* = $\{o_1\}$. *stack_full* becomes a method of o_1.

Rule 2. For a function f, if cardinality of *ref(f)* > 1 and cardinality of *modif(f)* = 1, then f becomes a method of the unique object in *modif(f)*.

This rule is motivated by the fact that conceptually we consider a function as a behavior of an object if it modifies its state. For example, *ref(stack_to_list)* = {*o1, o2*} and *modif(stack_to_list)* = {*o2*}, and *stack_to_list* becomes a method of o2. *stack_to_list* is a conversion function. In object-oriented programming, there are two possibilities to convert an object o1 into another object o2: (1) Ask o1 to become o2 (e.g., in smalltalk, method *asPolyline* in *Circle* class, which converts a circle into a polyline), and (2) create o2 from o1 (e.g., in smalltalk, method *fromDays* in *Date* class, which creates a date from an integer). With our approach the second solution is automatically taken. When available, an expert can make such a decision.

Rule 3. For a function f, if cardinality of *ref(f)* > 1 and cardinality of *modif(f)* > 1, then f must be sliced when possible to create a method for each object in *modif(f)*.

For example, *ref(global_init)* = {*o1, o2, o3*} and *modif(global_init)* = {*o1, o2, o3*}. *global_init* can be sliced to create three methods *init_stack, init_list, init_queue.* Actually, it is not always possible to break a function into cohesive methods. Other solutions can be used depending on the target OO language. In C++ for example, it is possible to define a function independently from any class. In other languages, a method can be associated to more than one class. Finally, it is possible to define a new object that aggregates the objects involved in *modif*(f), and put f as a method in that object.

CONCLUSION

Software maintenance is a complex and expensive task due to program understanding difficulties. We have addressed the issue of reverse engineering and reengineering through three important axes: program redocumentation, data and control flow analysis, and reengineering to OO technology. The main idea is centered on one strong hypothesis: Expertise, documentation, and developers of the application under maintenance are often not available and even when they are, their cost may be very high. Taking into account this reality, it is generally more efficient to choose an unsupervised approach. Such an approach is based on the source code, the only source of information judged reliable.

Unsupervised tools do not need domain expertise; they use, at most, heuristics to make the necessary decisions when identifying objects for example, and the results are not always reliable. Nevertheless, they are of a great help for maintaining legacy systems, by producing relevant abstractions. These abstractions allow us to have a wide set of solutions for both reverse and reengineering old systems.

BIBLIOGRAPHY

1. I. Sommerville, *Software Engineering,* 4th ed., Reading, MA: Addison-Wesley, 1992.
2. R. S. Pressman, *Software Engineering: A Practitioner's Approach,* New York: McGraw-Hill, 1987.
3. W. M. Ulrich, The evolutionary growth of software re-engineering and the decade ahead, *Amer. Program.,* **3** (10): 14–20, 1990.
4. R. K. Fjeldstad and W. T. Hamlen, Application program maintenance study: Report to our respondents, *Proc. GUIDE 48,* Philadelphia, 1979.
5. E. J. Chikofsky and J. H. Cross, Reverse engineering of software, *Encyclopedia of Software Engineering,* New York: Wiley, 1994, pp. 1077–1084.
6. M. M. Lehman and L. A. Belady, *Program Evolution,* New York: Academic Press, 1985.
7. A. Von Meyrhauser and A. M. Vans, Program Understanding—A Survey, CS94-120, Department of Computer Science, Colorado State Univ., August 1994.

8. D. J. Robson et al., Approaches to program comprehension, *J. Syst. Softw.,* **14**: 79–84, 1991.
9. S. Paul et al., *Theories and techniques of program understanding,* TR-74.069, IBM Canada Laboratory, October 1991.
10. A. Quilici, A memory-based approach to recognizing programming plans, *Commun. ACM,* **37** (5): 84–93, 1994.
11. E. J. Weyuker, The evaluation of program-based software test data adequacy criteria, *Commun. ACM,* **31** (6): 668–675, 1988.
12. F. G. Pagan, *Partial Computation and the Construction of Language Processors,* Englewood Cliffs, NJ: Prentice-Hall, 1991.
13. W. Kozaczynski and J. Q. Ning, SRE: A Knowledge-Based Environment for Large-Scale Software Re-engineering Activities, *ICSE '89,* Los Alamitos, CA: IEEE Computer Soc. Press, 1989, pp. 113–122.
14. W. Kozaczynski, S. Letovsky, and J. Q. Ning, A Knowledge-Based Approach for Software System Understanding, *KBSE '91,* Los Alamitos, CA: IEEE Computer Soc. Press, 1991, pp. 162–170.
15. J. Hartman, *Plans in software engineering—An overview,* Technical Report, AI Research Lab, The Ohio State Univ., 1995.
16. H. Lounis and W. L. Melo, Identifying and measuring coupling in modular systems, *8th Int. Conf. Softw. Technol. ICST '97,* Curitiba, 1997, pp. 23–40.
17. H. Lounis, H. A. Sahraoui, and W. L. Melo, Defining, measuring and using coupling metrics in object-oriented environment, SIGPLAN OOPSLA '97 Workshop on Object-Oriented Product Metrics, Atlanta, 1997.
18. S. S. Liu and N. Wilde, Identifying objects in a conventional procedural language: An example of data design recovery, *Conf. Softw. Maint.,* Los Alamitos, CA: IEEE Computer Soc. Press, 1990, pp. 266–271.
19. M. F. Dunn and J. C. Knight, Automating the detection of reusable parts in existing techniques, *Proc. Int. Conf. Softw. Eng.,* Los Alamitos, CA: IEEE Computer Soc. Press, 1993, pp. 381–390.
20. A. S. Yeh, D. R. Harris, and H. B. Reubenstein, Recovering abstract data types and object instances from a conventional procedural language, in L. Wills, P. Newcomb, and E. Chikovsky (eds.), *2nd Working Conf. Reverse Eng.,* Los Alamitos, CA: IEEE Computer Soc. Press, 1995, pp. 252–261.
21. S. Chen et al., A model for assembly program maintenance, *J. Softw. Maint. Res. Pract.,* **2**: 3–32, 1990.
22. M. T. Harandi and J. Q. Ning, Knowledge-based program analysis, *IEEE Softw.,* **7**: 74–81, 1990.
23. W. L. Johnson and E. Soloway, PROUST: KB program understanding, *IEEE Trans. Softw. Eng.,* **11**: 267–275, 1985.
24. R. A. Kemmerer and S. T. Eckmann, UNISEX: A UNIx-based Symbolic EXecutor for Pascal, *Softw. Pract. Exp.,* **15**: 439–458, 1985.
25. S. K. Abd-el-Hafiz, *A tool for understanding programs using functional specification abstraction,* Master's thesis, Univ. Maryland, College Park, MD, 1990.
26. S. Manke and F. N. Paulisch, *Graph Representation Language,* reference manual, University Karlsruhe, 1991.
27. A.V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools,* Reading, MA: Addison-Wesley, 1986.
28. M. Weiser, Program slicing, *IEEE Trans. Softw. Eng.,* **SE-10**: 352–357, 1984.
29. F. Lanubile and G. Visaggio, Function recovery based on program slicing, in D. Card (ed.), *ICSM '93,* Los Alamitos, CA: IEEE Computer Soc. Press, 1993, pp. 396–404.
30. K. B. Gallagher and J. R. Lyle, Using program slicing in software maintenance, *IEEE Trans. Softw. Eng.,* **17**: 751–761, 1991.
31. G. Canfora et al., Software-salvaging based on conditions, in H. A. Müller and M. Georges (eds.), *ICSM '94,* Los Alamitos, CA: IEEE Computer Soc. Press, 1994, pp. 424–433.
32. I. Jacobson and F. Lindstrom, Re-engineering of old systems to an object oriented architecture, *Proc. OOPSLA,* 1991, pp. 340–350.
33. H. C. Gall and R. R. Klösch, Finding objects in procedural programs, in L. Wills, P. Newcomb, and E. Chikovsky (eds.), *2nd Working Conf. Reverse Eng.,* Los Alamitos, CA: IEEE Computer Soc. Press, 1995, pp. 208–217.
34. J. Shin, *Migration of structured procedural C programs into object-oriented C++ based on code reuse,* Master's thesis, Univ. Pennsylvania, 1996.
35. R. M. Ogando, S. S. Yau, and N. Wilde, An object finder for program structure understanding, *J. Softw. Maint.,* **6** (5): 261–283, 1994.
36. P. E. Livadas and P. K. Roy, Program dependence analysis, *Conf. Softw. Maint.,* 1992, pp. 356–365.
37. D. Harris, H. Reubenstein, and A. S. Yeh, Recognizers for extracting architectural features from source code, in L. Wills, P. Newcomb, and E. Chikovsky (eds.), *2nd Working Conf. Reverse Eng.,* Los Alamitos, CA: IEEE Computer Soc. Press, 1995, pp. 252–261.
38. G. Canfora, A. Cimitile, and M. Munro, An improved algorithm for identifying objects in code, *Softw. Pract. Exp.,* **26** (1): 25–48, 1996.
39. M. Siff and T. Reps, Identifying modules via concept analysis, in M. J. Harrold and G. Visaggio (eds.), *Proc. ICSM '97,* 1997, pp. 170–179.
40. C. Lindig and G. Snelting, Assessing Modular Structure of Legacy Code Based on Mathematical Concept Analysis, in *Proc. Int. Conf. Softw. Eng.,* New York: ACM Press, 1997, pp. 349–359.
41. H. A. Sahraoui et al., Applying concept formation methods to object identification in procedural code, *Proc. IEEE Autom. Softw. Eng. Conf.,* 1997, pp. 210–218.
42. R. Godin et al., Applying concept formation methods to software reuse, *Int. J. Knowl. Eng. Softw. Eng.,* **5** (1): 119–142, 1995.
43. B. A. Davey and H. A. Priestley, *Introduction to Lattices and Order,* Cambridge, MA: Cambridge Univ. Press, 1992.
44. R. Godin, R. Missaoui, and H. Alaoui, Incremental concept formation algorithms based on Galois (concept) lattices, *Comput. Intell.,* **11** (2): 246–267, 1995.
45. H. Mili, On behavioral description in object-oriented modeling, *J. Syst. Softw.,* **34** (2): 105–121, 1996.

HAKIM LOUNIS
HOUARI A. SAHRAOUI
Centre de Recherche Informatique de Montréal (CRIM)
WALCÉLIO L. MELO
Oracle do Brasil and Universidade Católica de Brasília

SOFTWARE MAINTENANCE, REVERSE ENGINEERING. See SOFTWARE MAINTENANCE, REVERSE ENGINEERING AND REENGINEERING.