

COBOL/SRE: A COBOL System Renovation Environment

Andre Engberts, Wojtek Kozaczynski, Edy Liongosari, and Jim Q. Ning

Center for Strategic Technology Research, Andersen Consulting

100 South Wacker Drive, Chicago, Illinois 60606, U.S.A.

Abstract

COBOL/SRE is a software re-engineering and renovation environment for COBOL systems. It supports a wide range of features such as system-level analysis and browsing, program-level analysis and browsing, data model recovery, concept recognition, and code segmentation. COBOL/SRE is implemented on top of a distributed execution architecture to address issues of multi-user access, performance, and openness. This paper is an overview of the major features of COBOL/SRE and its underlying architecture.

1. Introduction

A current problem facing many large companies is how to deal with their legacy systems. These systems often supported critical business functions, but the cost of their maintenance and incremental functional enhancements becomes high. With the advent of new computing technologies and paradigms (LAN-based workstations, window-based graphical user interfaces, object-oriented languages, relational databases, distributed-computing techniques, etc.), there is growing demand to re-develop old systems. For example, a popular trend recently is to migrate mainframe-based systems to client/server architectures.

A majority of exiting software systems are COBOL-based. It is estimated that among 200 billion lines of existing code, 80% is COBOL [Fig90]. In many situations, old COBOL programs cannot be easily thrown away because they represent significant development and maintenance investment over the years. More importantly, they describe critical business rules that may not be accurately and explicitly documented anywhere else. Thus, there is a need for COBOL system re-engineering support.

COBOL/SRE is a renovation and re-engineering environment developed at the Andersen Consulting's Center for Strategic Technology Research (CSTaR) as a spin-off product of the software re-engineering research program.

COBOL/SRE supports a wide range of re-engineering features including system-level analysis and browsing, source code parsing, program-level analysis and browsing, data model recovery, concept recognition, and code segmentation. It executes on top of a distributed architecture to support group work, performance optimization, and openness.

This paper is a general overview of COBOL/SRE. Section 2 states the requirements and describes rationales behind the design of the environment. Major functionality of the environment and its underlying architecture are presented in sections 3 and 4, respectively. Section 5 reflects on how COBOL/SRE meets various re-engineering requirements stated in section 2. Section 6 serves as a summary of the paper.

2. Requirements

In order to determine COBOL/SRE's functionality, we must understand the areas or activities in software re-engineering where automated support is strongly desired. We have identified the following areas:

- ◆ *Dealing with source language variations.* The very first step in re-engineering is to process (or parse) the source code into some internal form to facilitate subsequent analysis. In the case of COBOL, the difficulty is that there are many dialects of the language. In addition, COBOL programs may include statements in other languages such as SQL calls or operating system directives. A re-engineering environment must be flexible enough to handle language variations.
- ◆ *Program-level viewing and browsing.* Program understanding is a major activity of almost any re-engineering tasks. The understanding can be greatly assisted by presenting various program views to the user. These views, which can be graphical or textual, should be interconnected so that the user can easily navigate among different views based on different relationships.

- ◆ *System-level viewing and browsing.* The user may need to understand the relationships among system programs and files. Views describing system objects should be presented to the user for browsing.
- ◆ *Automated understanding support.* Program understanding is a very difficult and time consuming task. It is always a plus for a re-engineering environment to provide the ability to automatically recognize instances of functional and data concepts.
- ◆ *Code quality analysis.* It is sometimes critical to be able to diagnose code quality in order to make decisions concerning appropriate re-engineering actions to take (i.e., whether to completely migrate the system to a new execution environment, to selectively reuse the code, or to abandon it altogether and develop new code.)
- ◆ *Code focusing.* This is another form of code understanding support. Old COBOL programs are typically large and complex. They are badly structured and often mix multiple functions in a single module. The user may want to focus on a set of program fragments (called a *code segment* or *slice*) that collectively implement an independent function.
- ◆ *Code factoring.* During system redevelopment, we want to throw away most of the old system. But some code segments may still be useful if they represent fundamental business functions. In order to make these code segments reusable, they must be extracted (*factored*) out of the old modules and packaged into stand-alone modules for easy reference by other modules.

In addition to the above areas, there are also less-functional but equally-important areas that a re-engineering environment must support:

- ◆ *Multi-user/group work support.* A re-engineering project is typically large and involves a group of analysts working together on a legacy system concurrently. It is not sufficient for a re-engineering environment to run on a stand-alone computer. Instead, it should be able to execute on a network of workstations and provide data sharing, security, and data persistency services.
- ◆ *Process support.* A re-engineering environment should not assume strong understanding of the re-engineering process from its users in order to be used effectively. Ideally, the process flow of the environment should be easy and intuitive to follow.
- ◆ *Performance and scalability.* Old systems may consist of hundreds of modules and hundreds of thousands of lines of code. A re-engineering

environment must be capable of dealing effectively and efficiently with very large systems.

- ◆ *Openness and adaptability.* Different re-engineering projects may have different requirements and objectives. A re-engineering environment should be designed in such a way that it is easy to customize, enhance, and bridge with external, commercial-off-the-shelf (COTS) tools.

The next section demonstrates how COBOL/SRE supports activities in the functional areas stated above. The following section (section 4) shows how the second group of requirements in the non-functional areas is addressed.

3. COBOL/SRE

A high-level view of COBOL/SRE is shown in Figure 1 in the next page. As shown in the figure, this environment advocates a re-engineering process model consisting of six steps: 1) system inventory, 2) system analysis, 3) system browsing, 4) program analysis, 5) program browsing, and 6) program segmentation. These steps are discussed below.

3.1. System Inventory

This step allows to load the source code files of a system (the system to be re-engineered) into COBOL/SRE. A system consists of several types of files, such as source programs, subprograms, copybooks, and JCL scripts. These files are represented within the environment as COBOL/SRE knowledge base objects. The user may create the system source file objects one by one, by interacting with COBOL/SRE's dialog windows. Alternatively, the user may provide a system description file to load and link the files into COBOL/SRE automatically in one pass. Such a file description specifies types and paths of all the files in the analyzed system.

3.2. System Analysis

This step captures the relationships among programs and data files in a system. Two types of relationships are captured: *program-to-program relationship* and *program-to-file relationship*. The program-to-program relationship represents both the module execution sequencing and the calling relationship. For batch systems the execution sequences can be recovered automatically by the analysis of JCL job steps. For on-line systems, however, this information must be provided by the user. This is because in most cases the control sequencing information of on-line systems is stored in the control tables of their transaction processing monitors and these tables are not easily

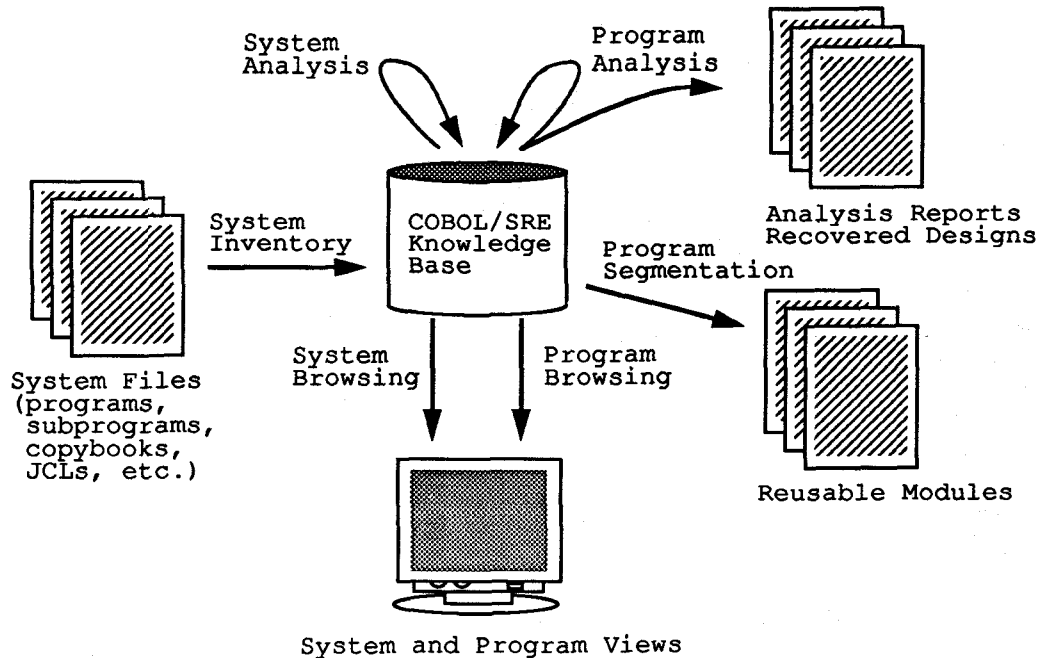


Figure 1. COBOL/SRE: a COBOL system renovation environment.

available. The calling relationships can be automatically recovered from the call statements (e.g., subprogram CALL statements, CICS LINK statements, etc.) present in the program source code.

The program-to-file relationship represents the system-level data flow. A program may read from a file and/or write into a file. A program may also create temporary files (by SORT and MERGE commands). For batch systems this relationship can be automatically recovered from program file definitions (FDs), file control descriptions (SELECTs), file manipulation statements (i.e., OPEN, READ, WRITE, SORT, MERGE, etc.), and JCL code. For on-line systems, transaction monitor tables must be analyzed.

3.3. System Browsing

The program-to-program relationship is presented in the form of a matrix (Figure 2a). In the matrix a cell labeled FOLLOWS/PRECEDES specifies that the execution of one program follows/precedes another. A cell labeled CALLS/CALLED BY specifies that a program calls/is called by another program.

Cells are active objects. If a FOLLOWS/PRECEDES cell is selected, the appropriate JCL file can be brought up in a text window. If a CALLS/CALLED BY cell is selected, a program browser window (described in Section 3.4) can be used to present the program section containing the corresponding CALL statement.

The program-to-file relationship is presented in another matrix (Figure 2b). In this matrix, R/W denotes READ/WRITE relationship while S/M indicates temporary files created by SORT/MERGE operation. Cells in this matrix are also active.

In addition to the matrix form, there are also graphical views. An example showing program-to-file relationship is given in Figure 3. In this view, programs are represented as program-listing icons, data files as pen-and-ink icons, and their relationships as directed lines.

3.4. Program Analysis

Syntactic and Semantic Analysis

This step computes and stores the syntactic and semantic information of the code in preparation for subsequent analysis and browsing activities. The step is divided into the following sub-steps:

- ◆ *Parsing.* The source code of a program is parsed into an internal knowledge base representation, called an *abstract syntax tree (AST)*.
- ◆ *Cross reference analysis.* Cross-reference links are established among data declarations, paragraph/section names, and their references.
- ◆ *Data layout analysis.* COBOL encourages data aliasing (through record fields, group items, redefinitions, and renamings). This analysis

dialects of COBOL the return addresses may be overridden or misused when the ranges of two PERFORM statements overlap. A standard complexity analysis based on McCabe and Halstead methods [McC76, HC87] is also performed in this step.

Data Model Recovery

The understanding and consolidation of data models is a primary objective of many COBOL system re-engineering projects. The data layout analysis only identifies the static memory mappings among the program data objects. The data model recovery goes a step further; it also recognizes *dynamic data mappings* caused by data assignments (MOVE, SET, or COMPUTE statement). The step generates a *virtual data model* and presents it in the form of a report. The report suggests how record structures and names should be consolidated and rationalized to make data declarations more consistent and comprehensive. The report may also reveal data anomalies, for example, moving data among records of different sizes and types.

Concept Recognition

Program understanding is recognized as the most time-consuming task in systems maintenance and re-engineering. Concept recognition is a knowledge-based technique that has been developed at CSTaR to automate the recognition of functional code patterns. The analyst is responsible for specifying the recognition knowledge encoded in the form of *plans*. Plans describe parts of concepts and constraints on and between them. A plan for recognizing an error-handling concept, for example, might have to specify two parts: (1) an assignment (MOVE or SET) to set an error flag, and (2) another assignment to send an error message to a screen field. Moreover, this plan needs to specify a constraint to require the two parts to appear on the same control flow path. More details about concept recognition can be found in [Nin89, KNS92].

3.5. Program Browsing

A majority of activities facilitating program understanding involve cross referencing, browsing, and analyzing various views of a program. In COBOL/SRE, these activities are supported within a program browser. The browser is a *syntax-directed editor* based on the AST representation of a program. A sample view of a program browser will be shown in a later part of this paper (Figure 6).

Cross Referencing

Using the browser the user can conveniently follow the cross-reference links between declarations and their references. If a variable reference is selected, for example, the user may use the Where Declared button under the

View option to localize the variable declaration.

Data Layout, Calling, and Control Flow Graphs

A data layout graph of data record WS-TRANS-RECORD is shown in the front window in Figure 4. Data objects in the graph can be traced back into the source browser by using the Where Declared menu option (background window in Figure 4). A call graph of a program describes PERFORM relations among paragraphs/sections (upper-right window in Figure 5). Selecting a node in the graph will highlight corresponding paragraph statements in the program browser (lower-right window in Figure 5). The control flow view of a node can be generated from the call graph window (upper-left window in Figure 5).

Concept Browsing

As we have described before, the concept recognition process identifies instances of programming and domain concepts in the code. The recognized concepts can be inspected in a source browse window. In order to see all instances of a concept recognized by a plan, the user must select the plan in a plan browse window. Code fragments corresponding to the recognized instances are highlighted in the program browser.

3.6. Program Segmentation

Old COBOL programs are often large and unstructured. Commonly, multiple functional units interleave each other inside a single module making them difficult to comprehend and reuse without breaking the module down into smaller, more functionally-cohesive units. Program segmentation is a re-engineering technique that developed at CSTaR to help user recognize, understand, document, and extract functionally-related code pieces [NEK93]. This technique enhances comprehensibility, maintainability, portability, and reusability of old code. There are two major steps in program segmentation: *focusing* and *factoring*.

3.6.1. Focusing

Let us call a piece of code consisting of program statements a *segment* or a *slice*. It is not necessary for statements in a meaningful segment to be localized (syntactically adjacent to each other), as long as the statements contribute to a common function. An example of a functional segment in a banking application would be sections of code implementing credit checking logic.

Focusing is a program understanding technique. It provides meaningful ways for the user to isolate statements that may not be syntactically adjacent but are semantically

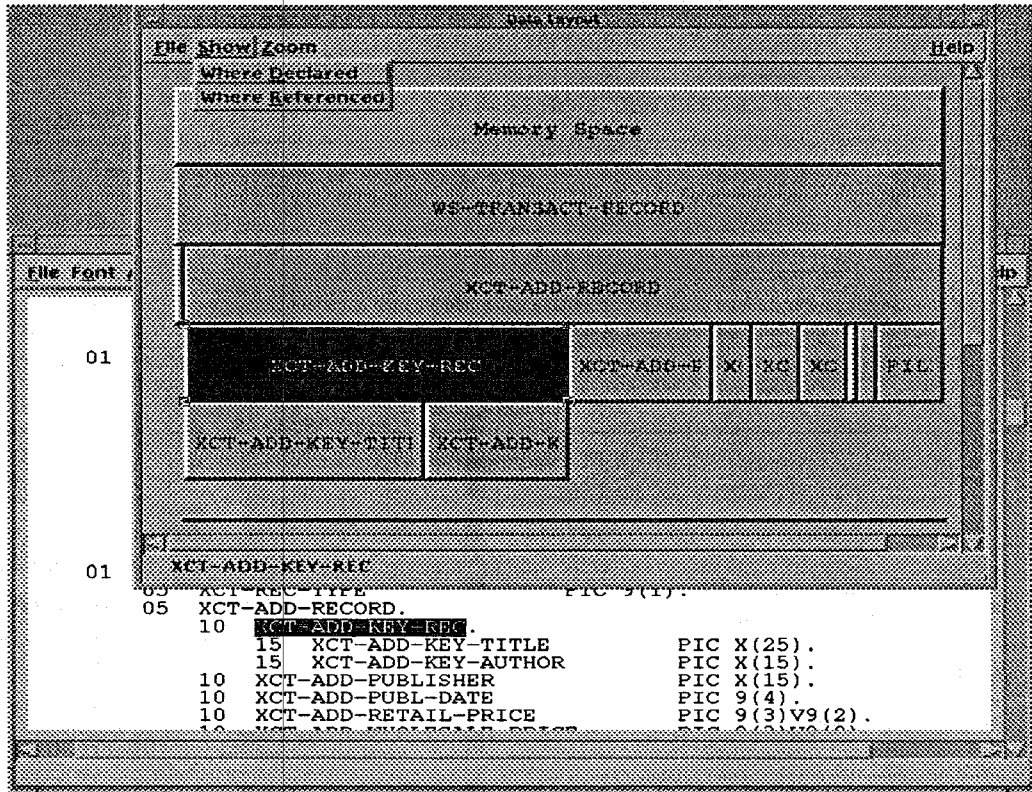


Figure 4. Data layout graph.

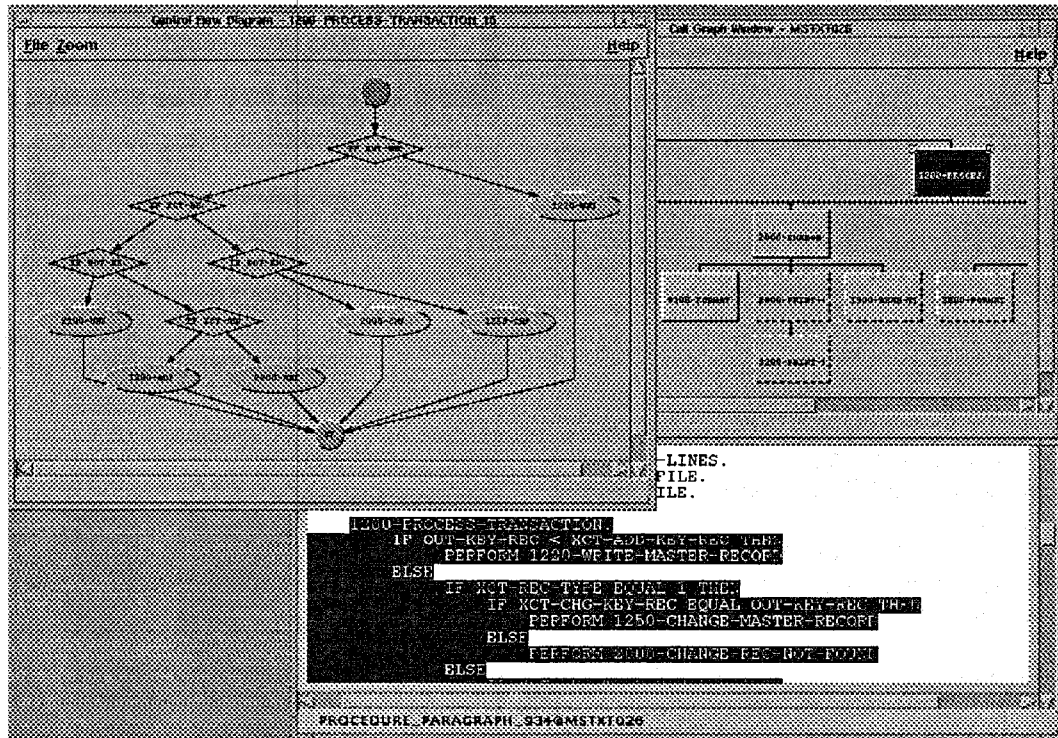


Figure 5. Call and control flow graphs.

related. There are five program focusing operations listed under the Focus button in the program browse window. They are described below.

Select Statements

This is the least constrained form of focusing designed to give the user maximum freedom in creating segments. The user uses simple mouse clicking to include arbitrary program statements into a segment. But the syntax-sensitive nature of the program browser guarantees that only complete statement objects can be selected. If the user attempts to select a non-statement object (e.g., a data division object), a warning message will be posted. The selected statements are highlighted (in reverse video) on the screen.

PERFORMed Statements

The program calling hierarchy usually reflects a functional decomposition of a program. Semantically, the PERFORM statement in COBOL is similar to the procedure

call statement in C or Pascal. Its target (called the PERFORM range) is one or a set of consecutive section(s) of code (called SECTIONS or PARAGRAPHS). The user may simply select a PERFORM statement in order to include all statements in the range of this PERFORM into a segment. If there are any PERFORM statements within the range, the process will continue to select statements in the ranges of these PERFORMs. All the selected statements are highlighted and put into a segment. Figure 6 shows the situation when 9 statements were automatically selected caused by the user selection of PERFORM 2800-PRINT-ONE-LINE.

Condition-Based Slicing

Business functions are sometimes structured along conditional tests. An automobile insurance application, for example, may discriminate among different policy calculation methods based on factors such as Age, Sex, Area, Auto Type, etc. It may be useful, therefore, to identify areas of a program reachable under a globally specified

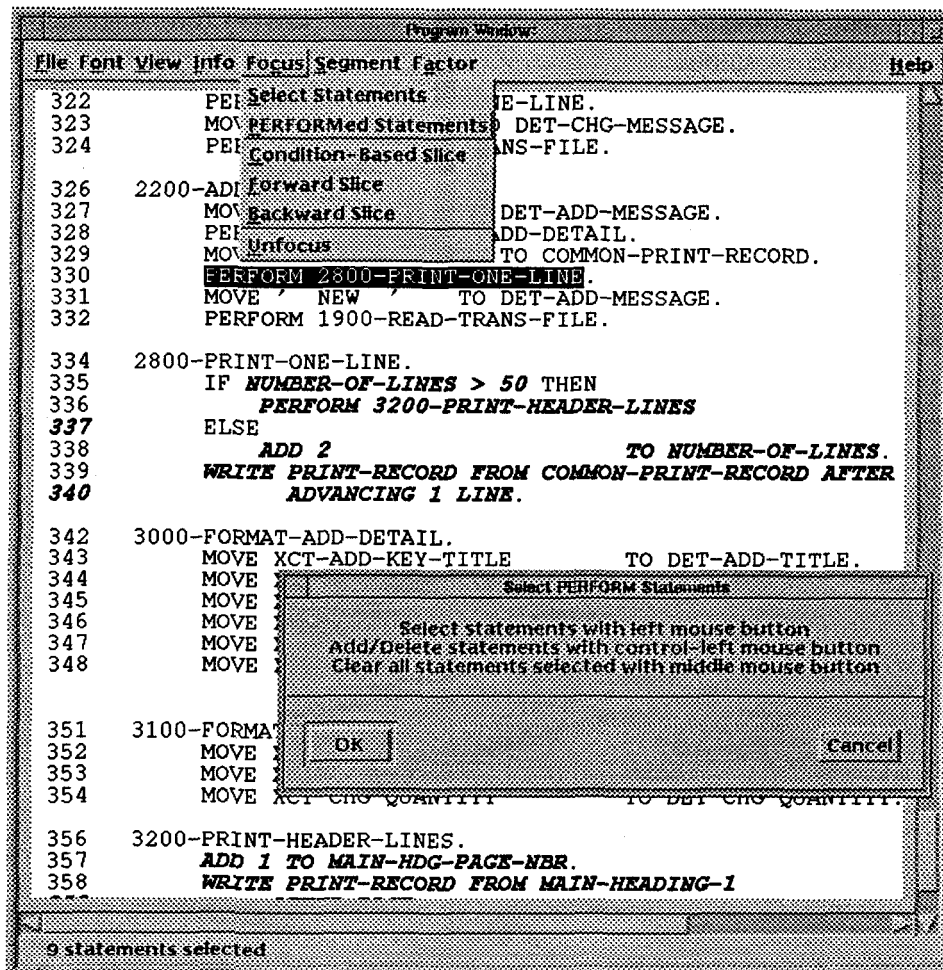


Figure 6. COBOL/SRE program browser and its focusing functions.

conditional expression such as:

STATE-ID = 'ILLINOIS' AND AGE > 30.

In condition-based slicing, the user specifies a logical expression and optionally a *slicing range* (in terms of where to start and end slicing in the program). Then, all the reachable program statements along control flow paths for which the given expression is true will be automatically included in a segment and highlighted.

Forward Slicing

Business functions often process input values. It is thus interesting to find out areas of code that depend on values of input variables.

Forward slicing (also called *ripple-effect analysis* [Liu88]) retrieves statements in the range that can be affected by a variable. A variable may affect a statement in terms of *data flow* [ASU87] if the statement uses a value of the variable. A variable may also affect a statement in terms of *control dependence* [FOW87] if whether the statement can be reached depends on the evaluation of the variable (normally, the variable is a part of a logical expression in a conditional branching statement). Forward

slicing is transitive in that when a statement is included in the slice all the variables set in this statement will be repeatedly used as new slicing variables.

Backward Slicing

Business functions normally produce results. It is thus interesting to find out areas of code that contribute to values of output variables.

Backward Slicing (or simply *slicing* [Wei84]) takes a variable and a slicing range and retrieves statements in the range that can effect the value of the variable. Again, a statement may affect a variable because of data flow or control dependence. Backward slicing process is also transitive.

3.6.2. Operations on Segments

The intention behind creating segments is to isolate functionally-related statements. Applications of individual focusing operations described above may not produce desired results. For example, in order to isolate the logic that reflects how an input variable affect an output variable, results from forward slicing and backward slicing may have to be combined. We have developed operations

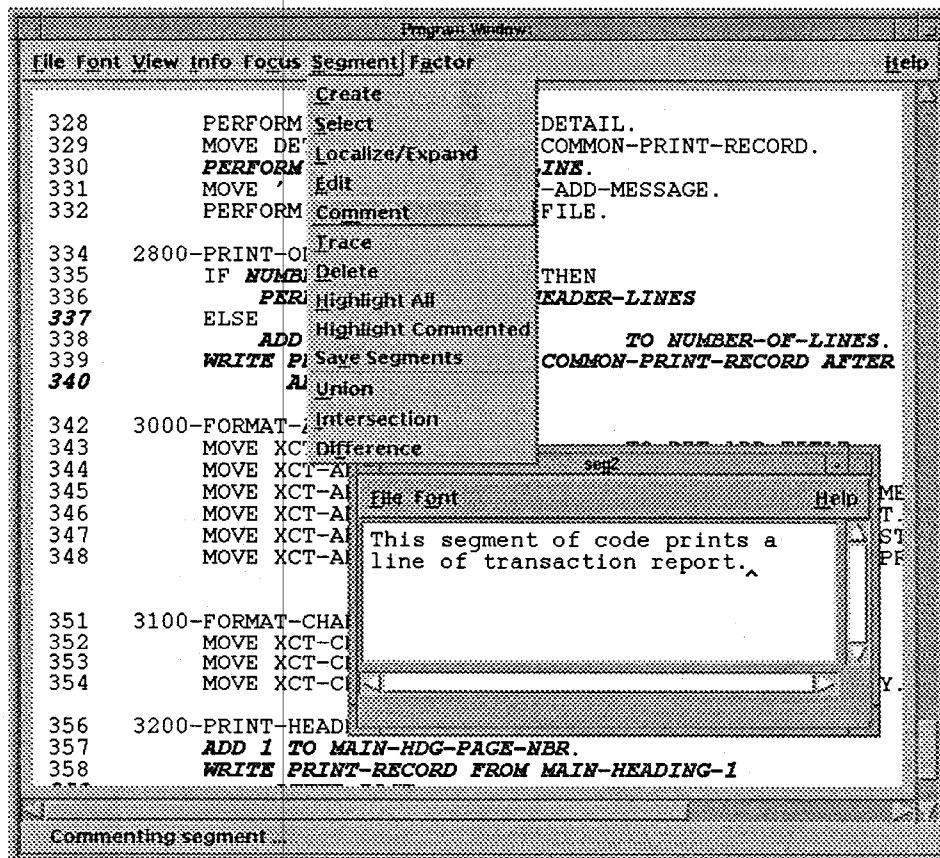


Figure 7. Segment management functions.

for managing and combining segments. These operations are listed under the Segment menu button and are shown in Figure 7.

Management Operations on Segments

The segment management operations help user understand, document, modify, delete, and save segments:

- Create - allows to name and store a segment.
- Select - retrieves a previously stored segment.
- Localize/Expand - extracts a segment into a localized region or expands it back into the program.
- Edit - adds or removes statements in a segment.
- Comment - attaches free-format explanation text to a segment (see Figure 7).
- Trace - helps navigate among statements in a segment.
- Delete - deletes a previously created segment.
- Highlight All/Highlight Commented - highlights statements in all segments/all commented segments.
- Save Segments - persistently saves currently created segments into the COBOL/SRE's database.

Set Operations on Segments

Set operations (Union, Intersection, and Difference) have been designed to support composition of segments. Each operation takes two segment objects as arguments and produces a new segment. For example, a segment

resulting from a Union includes all the statements (without duplications) in the two argument segments.

3.6.3. Factoring

Focusing localizes functional code regions and creates segments. Factoring, on the other hand, extracts code segments and packages them into independent COBOL subprograms ready for reuse. The factoring functions are listed under Factor button in the program browser. Details of how subprograms are generated are outside the scope of this paper due to the space limit. However, they can be found in [NEK93].

4. COBOL/SRE Execution Architecture

The underlying execution architecture of COBOL/SRE has been designed to meet the non-functional requirements stated in section 2. From the architecture point of view, COBOL/SRE is a collection of processes shown in Figure 8, that can run in a distributed, multi-workstation environment. These processes can be divided into the following two groups:

- ◆ *Function-support processes.* These are the processes that implement the COBOL/SRE functionality presented in the previous section.
- ◆ *Architecture-support processes.* These are the main subject of this section and are illustrated by shaded boxes in the figure.

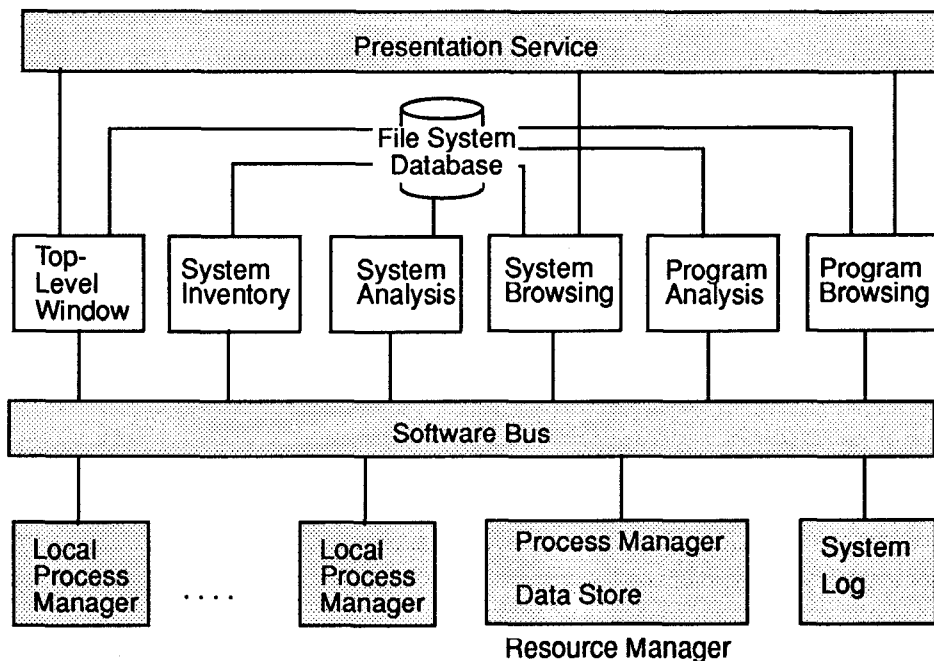


Figure 8. COBOL/SRE execution architecture.

The architecture is built around the *Software Bus* process [Rei90a]. The bus used in COBOL/SRE implements the *publish-subscribe* message distribution protocol and is based on the Brown University's bus also used in the FIELD system [Rei90b]. All COBOL/SRE processes send and receive messages exclusively through the bus. However, for performance reasons, large amounts of data are not sent in the form of messages but are stored and retrieved directly in/ from the database.

The second component of the architecture is the *Presentation Service* (also called *UI Server*). Presentation Service is a process that is a specialized layer between all processes that require communicating with the user and the X11 windowing environment. The process is internally composed of a number of modules that implement the following functions: 2D graphics editing and presentation, syntax-directed source code browsing, encapsulation of the standard Motif widgets, presentation (and editing) of data in the matrix form, and standard ASCII file editor.

The purpose of introducing the Presentation Service is two-fold:

- 1) From the application developers point of view, it raises the U/I primitives to a higher level of abstraction by encoding many of the standard look-and-feel of the interfaces.
- 2) From the architecture point of view, it allows many processes to communicate with the same Presentation Service or one process to communicate with many Presentation Service units. This in turn allows for: a) building an interface in which the user is not aware of where and how many processes are running, and b) two or more users to simultaneously operate (see each other's actions) on the same software object graphically presented to them by different Presentation Service processes.

The last critical component is the *Resource Manager*. Resource Manager is a process responsible for allocating and running other processes at workstations available to the environment. At least the following processes must run simultaneously for COBOL/SRE to work: the Bus, the Top Level Window, the Presentation Service, the Resource Manager, and as many Local Process Managers as the number of workstations used at the time.

Assume now, for example, that the user requests to perform parsing and basic analysis on a new program. This is done by the Program Analysis process that can run concurrently with all other processes on a workstation registered with COBOL/SRE. A workstation is registered with COBOL/SRE if there is a copy of the Local Process Manager running on it. A Local Process Manager monitors the activities of its workstation and reports them to the

Resource Manager. When the time comes to run the Program Analyzer, the Resource Manager does the following:

- ◆ analyzes the usage patterns of all workstations available (registered with) to COBOL/SRE
- ◆ selects one that has the most resources (most memory, least processes running, lowest usage level)
- ◆ sends a message to the Local Process Manager on the selected workstation to load and start the Program Analyzer

The Resource Manager uses a simple opportunistic strategy to allocate resources to processes. A number of copies of the Program Analyzer can run concurrently on a number of workstations and communicate with one or many Presentation Service processes (one or many users). Also, the same group of workstations can be allocated to multiple active instances of COBOL/SRE with their own sets of underlying processes. The process in one instance of COBOL/SRE are naturally grouped around its bus instance.

COBOL/SRE is written in C and runs under UNIX on Sun-4 and Sun-SPARC workstations. The modules of the function-support processes utilize the object management system from ProKappa [INT91]. The object management system was extended further internally to accommodate the locking mechanism in order for COBOL/SRE to work in a multi-user environment. The ProKappa interpretive and interactive environment was used to develop the code.

The Presentation Service has been written internally and it is based on Motif and X11. As we mentioned before, the bus has been taken from the Brown Workstation Environment libraries [PRB85] which are in public domain. The COBOL language parsers have been generated from the language specifications by a lexer/parser generator developed for CSTaR by Reasoning Systems Inc. and integrated with the ProKappa object management systems.

5. Meeting the Requirements

This section presents how and why the functional and architectural features of COBOL/SRE meet the requirements for software re-engineering support identified in section 2.

Dealing with Source Language Variations

The main parsing unit of the Program Analysis process has been designed to accept all major dialects of COBOL including: COBOL OSVS, COBOL VSC2, ANSI COBOL 74, ANSI COBOL 85, and COBOL II. The unit also calls parsers for SQL and CICS commands to cope with embedded, non-COBOL code. All the parsers were generated from specifications and can be easily modified/adapted.

Program-Level Viewing and Browsing

The COBOL/SRE supports many forms of program-level viewing and browsing including: syntax-directed program browsing, cross referencing, data layout graphs, calling graphs, and control flow graphs.

System-Level Viewing and Browsing

Understanding of the system-level execution flow is supported by browsing of the program-to-program relationship matrices and graphs. Understanding of the system-level data flow is supported by browsing of the program-to-file matrices and graphs. The cells in the matrices and objects in the graphs are active can be selected by mouse-clicking to view related program source code and JCL scripts.

Automated Understanding Support

Knowledge-based techniques are used to provide two forms of automated code understanding support: concept recognition and data model recovery. The concept recognition recognizes functional concepts implemented in the code and data model recovery recognizes data concepts.

Code Quality Analysis

COBOL/SRE identifies and reports control flow anomalies. It also performs McCabe and Halstead code complexity analysis.

Code Focusing

The user can conveniently slice or segment a program by selecting arbitrary statements, by selecting PERFORM ranges, and by slicing on logical conditions, and input and output variables.

Code Factoring

Code fragments can be extracted to form stand-alone modules suitable for reuse.

Multi-User and Group Work Support

The COBOL/SRE environment supports a group of analysts working on a system concurrently.

Process Support

A re-engineering process model has been encoded in COBOL/SRE. Under this process model a function (e.g. program parsing) cannot be invoked before its required preceding functions (e.g. system inventory) have been completed. This is controlled by presenting to the user only the menu options available at a given time. In some cases, COBOL/SRE will allow the user to invoke certain functions and will automatically decide additional func-

tions to be performed without user's intervention. For example, invocation of the control flow analysis will trigger execution of the cross referencing, if it has not been done so already.

Performance and Scalability

The COBOL/SRE's architecture has been developed primarily to meet the performance, scalability, and openness requirements. However, the extent to which we could distribute the COBOL/SRE functions has its limitations. The main limiting factor is the size of the complex objects that are processed. For example, knowledge base objects representing an AST of a 10 KLOC program is on the order of 10 MB. Moving such a large object into and from the address space of a process is an expensive (time consuming) operation. Therefore, many functions had to be packed into one application process to work in the same address space. For example, functions that have been separated from the Program Browser are the parser with all initial program analyzers, Plan Parser, and Concept Recognizer. This was done to allow for: (1) multiple programs to be parsed at the same time and (2) multiple concept libraries be matched against a program at the same time.

The performance of the environment in most areas is satisfactory. For example, parsing of COBOL programs and generation of their ASTs can be done at a speed of between 35 to 50 LOC per second. This is comparable with the speed of an average COBOL compiler running on the same workstation. Also, the behavior of the user interfaces is very much what the users of a high-end workstation would expect.

Openness and Adaptability

The openness has been achieved on two levels. First, due to its bus-centered architecture, COBOL/SRE can be easily integrated with other tools. Secondly, COBOL/SRE is itself an object-oriented environment whose parts are generated from specifications. Special care has been given to the design and configuration management so that COBOL/SRE's components are relatively easy to adapt.

6. Summary

The COBOL/SRE project has been a major undertaking. During its three years of research and development period more than ten people have been directly involved in it. The current version of COBOL/SRE consists of more than 300 source files containing over 240 KLOC. These source code files were written in a number of languages including C, ProKappa C, C++, a domain modeling language, and lexer/parser specification languages. The COBOL/SRE environment is able to accept and process

input files written in 9 different languages: COBOL, CICS, SQL, JCL, plan language (for concept recognition), domain modeling language, and a system file description language (for system inventory).

The COBOL/SRE represents a major advance in software re-engineering and renovation technology. It provides many unique features that are not available in similar re-engineering tools such as VIA/Renaissance™ [VIA91] developed and marketed by VIASOFT and REFINE/COBOL™ developed by Reasoning Systems Inc. The unique features of COBOL/SRE are summarized below:

- ◆ *system-level analysis and browsing*
- ◆ *full parsing capability*
- ◆ *complete program control flow, data flow, and control dependence analysis capability*
- ◆ *data model recovery*
- ◆ *knowledge-based program concept recognition*
- ◆ *program focusing based on a variety of slicing techniques*
- ◆ *intelligent program factoring*
- ◆ *distributed execution architecture supporting group work, performance optimization, openness, and adaptability*

Currently COBOL/SRE is being tested on real-life re-engineering and renovation projects to evaluate its practical utility and performance.

Acknowledgment

The authors would like to acknowledge valuable contributions of Laura Bell, Carmen Checa, Jeff Leane, Dan Marks, Tor Mesoy, Jim Miller, Tom Sarver, and Jeff West, who were members of the COBOL/SRE management and development team.

References

- [ASU87] Aho, A. V., Sethi R., Ullman J. D., *Compilers Principles, Techniques, and Tools*, Addison-Wesley, 1987.
- [FOW87] Ferrante, J., Ottenstein, K., and Warren, J., "The Program Dependence Graph and its Use in Optimization," *ACM Trans. on Programming Languages and Systems*, 9(3), July 1987, pp. 319-349.
- [Fig90] Figlibuo, R., "Re-engineering, De-engineering, and Over-engineering," *Seventh International Conference and Exposition on Software Maintenance and Re-engineering*, Washington, D.C., April 1990.
- [HC87] Harrison, W., and Cook, C., "A Micro/Macro Measure of Software Complexity," *Journal of Systems and Software*, 7(1987), pp. 213-219.
- [INT91] INTELLICORP, INC., "ProKappa User's Guide," August 1991.
- [KLN91] Kozaczynski, W., Letovsky, S., and Ning, J., "A Knowledge-Based Approach to Software System Understanding," *Sixth Knowledge-Based Software Engineering Conference*, Syracuse, New York, September 1991.
- [KNS92] Kozaczynski, W., Ning, J., and Sarver, T., "Program Concept Recognition," *7th Knowledge-Based Software Engineering Conference*, McLean, Virginia, September 1992.
- [Liu88] Liu, S. S., "Some Approaches to Logical Ripple Effect Analysis," *Technical Report*, Software Engineering Research Center, University of Florida, October 1988.
- [McC76], McCabe, T. J., "A Complexity Measure," *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 4, December 1976, pp. 308-320.
- [Nin89] Ning, J. Q., *A Knowledge-Based Approach to Automatic Program Analysis*, Ph.D. thesis, University of Illinois at Urbana-Campaign, October 1989.
- [NEK93] Ning, J. Q., Engberts, A., and Kozaczynski, W., "Recovering Reusable Components from Legacy Systems by Program Segmentation," *Working Conference on Reverse Engineering, ICSE-15*, Baltimore, Maryland, May 1993.
- [PRB85] Pato, J. M., Reiss, S. P., and Brown, M. K., "An Environment for Workstations," *IEEE Conference on Software Tools*, New York, 1985.
- [Rei90a] Reiss, S. P., "Connecting Tools Using Message Passing in the Field Environment," *IEEE Software*, July 1990, pp. 57-66.
- [Rei90b] Reiss, S. P., "Interacting with the FIELD environment," *Software - Practice and Experience*, Vol. 20 (S1), June 1990, pp. 89-115.
- [VIA91] VIASOFT, "VIA/Renaissance™ - Integrated Re-engineering for COBOL Programs," 1991.
- [Wei84] Weiser, M., "Program Slicing," *IEEE Transaction on Software Engineering*, Vol. SE-10, No. 4, 352-357, July 1984.