Rapid System Understanding: Two COBOL Case Studies

Arie van Deursen, Tobias Kuipers*
CWI, P.O. Box 94079
1090 GB Amsterdam, The Netherlands
http://www.cwi.nl/~{arie,kuipers}/, {arie,kuipers}@cwi.nl

Abstract

Rapid system understanding is required in the planning, feasibility assessment and cost estimating phases of a system renovation project. In this paper, we apply a number of analyses on two large legacy COBOL systems from the banking field. We describe the analyses performed, and discuss possible interpretations of these analyses. Lessons learned include: (1) The open architecture adopted is satisfactory, and can take advantage of a wide range of understanding tools available; and (2) To handle inter-system variability effectively, the flexibility of lexical analysis is required.

1991 Computing Reviews Classification System: D.2.2, D.2.7., D.3.4.

Keywords and Phrases: Software visualization, lexical analysis, software reuse.

Note: To appear in *Proceedings of the 6th IEEE International Workshop on Program Comprehenson*, June, 1998, Ischia.

Note: Work carried out under project SEN-1.1, *Software Renovation*.

1 Introduction

Rapid system understanding is the process of acquiring understanding of a legacy software system in a short period of time. Typical tasks that require rapid system understanding are:

- Assessing the costs involved in carrying out a European Single Currency or year 2000 conversion;
- Estimating the maintainability of a system, for example when deciding about accepting or issuing a maintenance outsourcing contract;
- Investigating the costs and benefits of migrating a system to an object-oriented language, in order to increase its flexibility and maintainability;
- Determining whether legacy code contains potentially reusable code or functionality.

Performing these tasks should be cheap: one expects a cost estimate of, say, a year 2000 conversion to be significantly less expensive than carrying out that conversion. This is where rapid system understanding differs from more traditional system understanding. Accepting a less detailed understanding and slightly inaccurate results, a first assessment can be made quickly.

We assume that the engineer who needs to acquire understanding of a legacy system has negligible previous experience with it. He may be unfamiliar with some of the languages or dialects used in the legacy code. The systems involved are typically large, multi language, over 10 years old, and written by different programmers.

In this paper, we take two 100 KLOC COBOL systems from the banking area as our starting point. We address a number of related questions: What tools or techniques can be used in rapid system understanding? How well do they work for our case studies? What information can be extracted from legacy source code, and how should this information be interpreted?

The paper is organized as follows. In Section 2 we explain what tools and techniques can be used, and how these cooperate. In Section 3 we list the characteristics of the two COBOL systems under study. In Section 4 we describe the kind of information we extracted from the legacy code, while in Section 5 we discuss the possible interpretation of this data. In Sections 6, 7 and 8 we summarize related work, conclusions and future work.

*This work was sponsored in part by bank ABN AMRO, software house Roccade, and the Dutch *Ministerie van Economische Zaken* (Department of Commerce) via SENTER Project #ITU95017 "SOS Resolver". The authors would like to thank the members of the *Migrating COBOL to Object-Oriented COBOL* Resolver task group: Hans Bosma, Erwin Fielt, Jan-Willem Hubbers, and Theo Wiggerts.

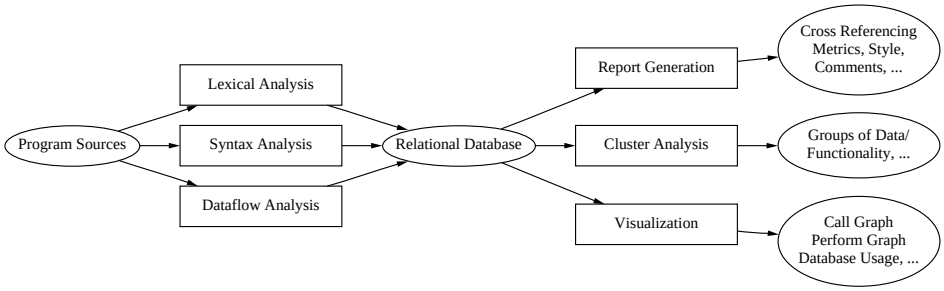

Figure 1. Architecture of tool set used

2 Tool Architecture

Rapid system understanding implies summarizing of data. In order to understand a large legacy system, it is necessary to quickly find the "essence" of such a system. What constitutes this essence largely depends on the reasons for trying to understand the system.

Our approach is to analyze the code using generic tools that have no a-priori knowledge of the system. The results of this analysis are then fed into a central repository. In turn, this repository can then be queried, printed, visualized, etc. The querying of the repository leads to a certain degree of understanding of the system. We can exploit this understanding by creating analysis techniques that do contain (a degree of) specific knowledge of the system. This will lead to data in the repository that is more suited for our specific goals. Again, this data can be visualized, queried, etc., to gain a greater understanding of the system. This process is repeated until the engineer who tries to understand the system has gained sufficient knowledge of it.

The general architecture of our tool set consists of three main parts, as shown in Figure 1. The first part is the code analysis part, the second the repository, and the third the tools that manipulate and present data from the repository.

For the code analysis part lexical, syntactic or other forms of analysis can be used. The figure distinguishes lexical, syntactic [2], and data flow analysis. For the purpose of rapid system understanding, it will generally suffice to use lexical analysis. It can be performed faster than syntactic analysis, and is much more flexible [16].

To extract a number of relations from COBOL legacy systems, we have developed a simple Perl [18] script called `recover`. It knows about COBOL's comment conventions, keywords, sectioning, etc. It can be used to search the sources for certain regular expressions, and to fill tables with various relations, for example pertaining to the usage of databases, call structure, variable usage, etc. The data extracted for COBOL is discussed in full detail in Section 4.

We store the analysis results as comma-separated-value (CSV) files. Such files can be easily queried and manipulated by Unix tools such as `awk` [1] and `join`, and can be read by arbitrary relational database packages enabling us to use SQL for querying the data extracted from the sources. These tools can also be used to generate reports, for example on the usage frequency of certain variables, or containing the fan-in/fan-out metric of sections of code.

Many relations stored in the repository are graphs. We use the graph drawing package `dot` [9] for visualizing these relations.

3 Cases Investigated

Central in our research are two COBOL systems from the banking area, which in this paper we will refer to as Mortgage and Share. The respective owners of these systems are in general satisfied with their functionality, but less satisfied with their platform dependency. They are interested in extracting the essential functionality of these systems, in order to incorporate it into a more flexible, object-oriented, architecture. Thus, questions of interest include: Do these systems contain reusable code? What fraction of the code is platform specific? Which data fields represent business entities? Which procedures or statements describe business rules?

The sizes of the two systems are summarized in Figure 2. Mortgage is a COBOL/CICS[1] application using VSAM[2] files. It is partly on-line (interactive), partly batch-oriented, and in fact only a subsystem of a larger (1 MLOC) system. Share is an IMS[3] application which uses both DL/I[4] (for accessing an IMS hierarchical database) and SQL (for accessing DB2 databases).

For Mortgage, we had system-specific documentation available, explaining the architecture and the main functionality of the programs. The documentation marked several programs as "obsolete": some of these were included in the

[1] CICS is Customer Information Control System, a user interface and communications layer

[2] VSAM is Virtual Storage Access Method, an access method for records

[3] IMS is Information Management System, a database and data communication system

[4] DL/I is Data Language 1, a database management language

Mortgage	no	LOC	avg
copybooks	1103	49385	44
programs	184	58595	318
total	1288	107980	83

Share	no	LOC	avg
copybooks	391	16728	42
programs	87	104507	1201
total	479	121235	253

Figure 2. System inventory.

version distributed to us, however. For Share, no specific documentation was available: we only had a general "style guide" explaining, for example, the naming conventions to be used for all software developed at the owner's site.

4 Collected Data

In this section, we discuss how we used the tool set of Section 2 to extract data from the Mortgage and Share sources, and how we displayed this data in a comprehensible manner. The results of the analysis will be discussed in Section 5.

System inventory The system inventory table summarizes available files, sizes, types (copybook, program), and languages used (COBOL, CICS, SQL, DL/I, ...). The copybook table indicates how copybooks are included by programs (a simple lexical search for the arguments of the COPY command). If appropriate, for certain files (copybooks) it can be detected that they were generated, for example if they contain certain types of comment or keywords. The system inventory derived for Mortgage and Share was used to obtain Figure 2.

Program call graph The call relation is a first step in understanding the dependencies between the programs in Mortgage and Share. Deriving the call graph for COBOL programs (see Figure 3 for the batch call graph of Mortgage) is not entirely trivial. First of all, the argument of a CALL statement can be a variable holding a string value, i.e., it can be dynamically computed. The most desirable solution to this problem is to have some form of constant propagation. In our case, for Mortgage it was sufficient to search for the values of certain variables or, in Share, for strings matching a certain lexical pattern.

In Share, we encountered further complications. Rather than a regular CALL statement, each call is in fact a call to some assembler utility. One of the arguments is a string encoding the name of the program to be called, as well as the way in which that program is to be loaded. The assembler routine subsequently takes care of loading the most recent

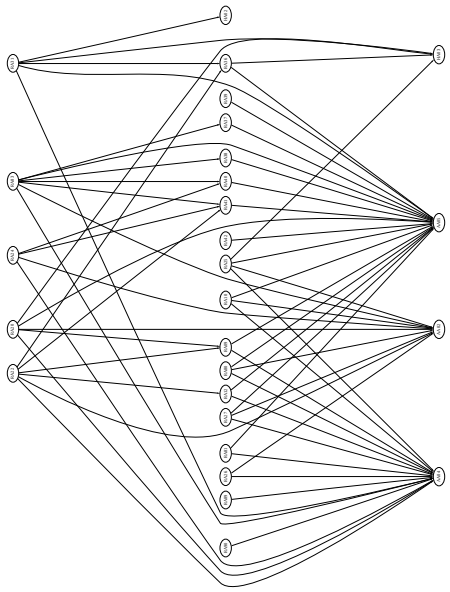

Figure 3. CALL graph for the batch part of Mortgage**.**

version of this program. Once we understood this mechanism, it was relatively easy to derive the call graph using lexical pattern matching.

In Mortgage, the use of CICS provides further possibilities of calling programs. The first is the CICS LINK statement, which is similar to a CALL statement. The second is the CICS XCTL statement. This takes care of invoking a program just before or after an end-user has filled in a screen as presented in an on-line session. In Mortgage, the XCTL calls could be extracted by tracing the value of a specific variable.

Observe that these special call conventions imply that commercial reengineering tools should be sufficiently flexible to allow such organization-specific extensions. We have looked at two of the most advanced COBOL reengineering tools, Reasoning/COBOL [11] and MicroFocus/Revolve [14]. Both support call graph extraction from abstract syntax trees, but neither is able to produce the on-line call graph of Mortgage or the batch call graph of Share. They can be adapted to produce these graphs, but that will be more time consuming than specifying a simple lexical search, making the latter option more attractive in a rapid system understanding setting.

Database usage A viable starting point for locating data entities of interest is the data that the system reads from or stores in persistent databases. In Mortgage, VSAM files are used, and both COBOL as well as CICS constructs to access them. In Share, VSAM, hierarchical IMS and relational DB2 tables are used, and COBOL I/O statements,

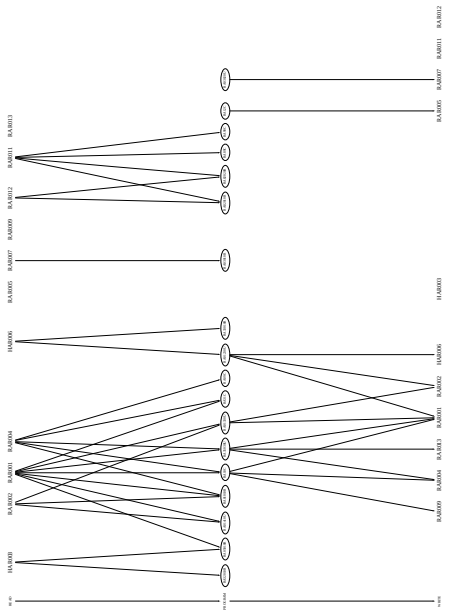

Figure 4. Graphical representation of the CRUD matrix of Mortgage**.**

SQL and DL/I to access them.

In an SQL system, a datamodel listing all tables with their field names and types is explicitly available. The `recover` tool can be used to extract this model from the source. In a non-SQL application such as Mortgage, this datamodel is not available. What can be derived, though, is the COBOL record definition used when writing to or reading from files.

Share uses 36 different tables, with in all 146 different field names. The number of fields per table varies from 1 to 40 – suggesting that many tables share certain fields. To make this visible we generated a 60-page LaTeX document. For each table we have one section listing the fields and their types, as well as the programs in which the table was declared. We then used `makeindex` to generate an index, indicating at what pages the tables, fields, types, and programs were used.

The CRUD — create, read, update, delete — matrix indicates how databases are manipulated by programs. As viewing a CRUD matrix of a large system is cumbersome, we use the graphical representation of Figure 4. The left-hand column contains records read, the right-hand one records written, and the middle column lists the programs involved. An arrow from a record to a program indicates a read, and an arrow from a program to a record indicates a write.

Field usage The database usage and datamodel provide a rough overview of the database operations per program. In many cases, it is useful to look at the table field level as well.

Program	*Section*	*# vars*
RA01330U	B200-UPDATE-UIT-FIB	35
RA01230U	B200-UPDATE-UIT-FIB	35
RA01010R	C100-VUL-FIB	34
RA31C	R300-MBV-RELATIENR	32
RA31	R300-MBV-RELATIENR	32
RA20	R220-VERWERK-30-31	26
RA20	R210-VERWERK-20-21	25
RA20	R200-VERWERK10	25
RA35010R	B300-VALIDATIE	16
RA33010R	B300-VALIDATIE	16
RA20	R500-SPAARHYPOTHEEK	13
...	...	...

Figure 5. Number of record fields used per section.

In order to expose shared use of database fields, we collect all occurrences of database field identifiers per section per program. From this table, we derive a list as shown in Figure 5. In Mortgage essentially 35 datafields are contained in one large table. Figure 5 shows how many of these datafields are used in each section. Of particular interest are those sections dealing with a small number (less than, say, 10) of data fields only.

We extracted the field usage relation using lexical analysis only. From the database declarations we extracted the field names. We then matched on section declarations to find section boundaries, and identified a field usage if one of the lines in a section contains a field name as substring. Clearly, this is an approximative method, relying on systematic naming conventions: the more accurate way would be to parse the source and do dataflow analysis to follow field usage through the code. For the two cases studied, though, the results obtained by lexical analysis were sufficiently accurate, at least for rapid system understanding purposes.

Section call graph A last relation we experimented with was the call graph at the section and paragraph level (the PERFORM graph). The perform relation can be used to visualize the calling structure of sections in individual programs. Since there are many programs, this is only useful if a selection of interesting programs is made beforehand (e.g. using the CALL graph).

Apart from visualizing the perform graph per program, all sections of the system can also be listed and sorted according to some metric. Figure 6 shows all sections with a fan-in of at least 3, and a fan-out of at most 3. It can be used to search for sections with an attractive fan-in/fan-out ratio.

At the system level, the sections included in copybooks are of particular interest. These sections were designed to be reusable, and therefore are natural candidates for further reuse. Figure 7 lists some of these sections for Mortgage, together with the number of other programs and sections containing a call to that section (fan-in).

Program	*Section*	*Fan-in*	*Fan-out*
RA20	R320-LEES-HAB006	23	1
RA20	R330-DATUM	16	2
RA83	R995-CLOSE-FILES	12	0
RA22	R995-CLOSE-FILES	12	0
RA80	R30-PRINT-REGEL	10	0
RA23	R995-CLOSE-FILES	9	0
RA20	R995-CLOSE-FILES	8	0
RA24	R995-CLOSE-FILES	7	0
RA80	R70-WRITE-RAB011S	5	0
RA38	R300-VERTAAL-GROOT-KLEIN	5	0
...	...	...	...
RA26	R995-CLOSE-FILES	3	0
RA23	R60-REDSEQ-RAB008	3	1
RA20	R212-VUL-OUD	3	3

Figure 6. Sections sorted by fan-in and fan-out.

Section	*Performed*	*Sections*	*Programs*
Y800-PRINT	145	17	12
Y998-HANDLE-ERROR	92	92	92
Y010-40-AIB	89	89	89
Y010-00-AIB	81	81	81
Y502-MASKER	80	25	25
Y020-00-FIB	79	79	79
Y020-40-FIB	66	66	66
...	...	...	...
Z610-82-RAB011	7	5	4
Z610-80-RAB011	7	6	5
Y675-STD-STRAAT	7	4	2
Y625-ELF-PROEF	7	6	6
Y415-INFO-SCHONEN-NIT	7	3	3
Z610-03-RAB011	6	4	4
Y750-STD-NAAM	6	5	3

Figure 7. Sections performed by different sections and programs.

Further experiments While finding out how to extract the various relations from the sources, we also used `recover` as an enhanced COBOL lexer. The `recover` script contains several functions to ignore COBOL comment columns and lines, to split lines into strings, numbers, identifiers, keywords, and other tokens, to search for arguments of keywords, to expand copybooks, to record information encountered in earlier lines, and to store results into tables. These functions were fruitfully used to acquire an understanding of conventions used, relevant data, etc.

5 Interpreting Analysis Results

In this section, we discuss how the graphs and reports derived in the previous section helped us to actually understand the two COBOL systems under study.

Understanding copybooks Rapid system understanding is a mixture between looking at global system information like the call graph and looking in more detail at a specific program in order to obtain a feeling of its particularities. One of the aims during rapid system understanding is to reduce the number of programs that need to be studied in detail. Having the copybook relation explicitly available will help to avoid looking at copybooks that are in fact never included.

For Share, 136 of the 391 (35%) copybooks were not used; for Mortgage 673 of the 1103 (61%) were not used. These large numbers can partly be explained as Mortgage is part of a larger system: for safety, all copybooks were included. Likewise, Share relies on general utilities used at the owner's site; to be safe many of these were included. We have not yet looked at other forms of dead code, such as sections or programs never called. To detect the latter, one should also have all JCL[5] scripts available, which we did not have for our case studies.

Another use of the copybook relation is to identify patterns in the copybook inclusions. It turned out, for example, that the batch and the on-line part of Mortgage use two almost disjoint sets of copybooks.

Call graph and reusability The batch call graph for a part of Mortgage is shown in Figure 3. This graph shows particularly well that we can identify:

- Programs with a high fan-out. From inspection we know that these are typically "control" modules. They invoke a number of other programs in the appropriate order. In Figure 3, they are mostly grouped in the left-hand column.

[5] JCL is Job Control Language, a shell-like system for MVS.

- Programs with a very high fan-in, i.e., called by most other programs. These typically deal with technical issues, such as error handling. From inspection it is clear that they are tightly connected to legacy architecture, and are not likely to contain reusable code. In Figure 3, they are grouped in the right-hand column.
- Programs with a fan-in higher than their fan-out, yet below a certain threshold. These programs can be expected to contain code that is reusable by different programs. In Figure 3, they are mostly in the middle column. These programs form the starting point when searching for candidate methods when trying to extract an object-oriented redesign from legacy code.

For the batch part of Mortgage, this categorization worked remarkably well.

The call graph based on CICS LINK commands (not shown) contains the remaining calls. Due to the presence of error handling modules, this call graph was difficult to understand. Removing all modules with fan-in higher than a certain threshold (say 10), we obtained a comprehensible layout.

For Mortgage, this analysis of the call graph led to the identification of 20 potentially reusable programs that performed a well-defined, relatively small task.

For Share, only 50% of the programs were contained in the call graph; the remaining programs are called by JCL scripts, which we did not have available. Therefore, for Share further ways of identifying reusable code will be required.

At a finer granularity, analysis of the PERFORM graph will be an option. In principle, the same categorization in control, technical, and potentially reusable code can be made. The derived table of Figure 6 can help to find sections of an acceptable fan-in. Clearly, this list is rather large, and not all sections will be relevant: we decide to inspect the code of a section based on its name. For example, we could probably ignore CLOSE-FILES, but should take a look at VERTAAL-GROOT-KLEIN[6], especially since this section occurs in three different programs.

Analysis of the sections included in copybooks as shown in Figure 7 will proceed along the same lines: based on the number of perform statements and the name of the section (for example, STD-STRAAT, indicating processing of a STRAAT, i.e., street), we will inspect code of interest. Surprisingly, Share does not use any sections defined in copybooks.

Understanding data usage The tools provide three ways to understand the data usage. The first is the index of tables, attributes, types and programs derived from SQL data definitions. This index can be used to detect, for example, sets of attributes that occur in several different tables and hence may be (foreign) keys.

The second aid is the CRUD matrix, which shows which programs read or write certain tables. We used Figure 4, which shows the database usage for Mortgage, to identify databases that are only read from (for example the "zip-code book", shown in the top-left corner of Figure 4) or only written to (logging, shown in in the top-right corner), databases used by many programs, or programs using many databases. We also used this figure to identify those databases that are used by most other programs. For Mortgage, there are only three such databases. The tools indicate which (level 01 COBOL) record definitions are used to access these, and the fields in these records we considered as the essential business data of Mortgage.

The third possibility aims at finding out how these data fields are used throughout the system, using Figure 5. It can help to group data fields based on their shared use, or to locate sections of interest, potentially containing core business functionality. Again, we will use this list to select sections of interest manually. Examples are sections called VALIDATIE, which contain code for checking the validity of fields entered via screens.

Reusability assessment In the preceding sections, we have discussed how certain information can be extracted from COBOL sources using lexical analysis methods (Section 4) and how we can use this information to understand the legacy system at hand (Section 5). Does this acquired understanding help us to answer the questions posed in Section 3?

- *Do the systems contain reusable code?* Based on the call graph several programs and sections were identified which, after inspection, turned out to contain well-isolated reusable functionality.
- *Which data fields represent business entities?* The tools help to identify those data fields that are written to file and used by most programs: the indexed representation helps to browse these fields and filter out certain fields that are perceived as "non-business".
- *Which statements describe business rules?* An inventory of the data fields used is made per section: those dealing with several fields are likely to describe business-oriented procedures.
- *What fraction of code is platform-specific?* Of the 340 sections of Share 177 (approximately 50%), refer to at least one data field. Thus, an initial estimate is that the other 50% is likely to contain platform-specific code. For Mortgage, 510 of the 2841 sections (only 18%)

[6] Dutch for *map-upper-lower*

refer to the data items stored on file. Thus, 82% appears to be platform-oriented rather than business oriented. Inspection of the functionality shows that this is the case: a large part of Mortgage deals with CICS-specific details (implementing a layer on top of CICS).

Evidently the answers to these questions are approximative. If a full reuse, reengineering, or re-implementation project is to be started, this project will require a more detailed answer to these questions. In order to decide to embark upon such a project, fast answers, such as those discussed in this section, obtained at minimal costs, are required.

6 Related Work

Lexical analysis of legacy systems Murphy and Notkin describe an approach for the fast extraction of source models using lexical analysis [16]. This approach can be used for the "analysis" phase (as showed in Figure 1), in stead of `recover`. Murphy and Notkin define an intermediate language to express lexical queries. Queries composed in this language are generally short and concise. Unfortunately, the tool was not available, so we were not able to use this tool for our COBOL experiments.

An interesting mixture between the lexical approach of AWK and matching in the abstract syntax tree is provided by the TAWK language [10]. Since TAWK is not (yet) instantiated with a COBOL grammar, however, we could not use it for our experiments.

Reengineering tools There are a number of commercial reengineering tools that can analyze legacy systems, e.g. [14, 11]. They are either language-specific (mainly COBOL), or otherwise based on lexical analysis. Lexical analysis provides a level of language independence here, and makes the system easier to adapt to new languages and dialects.

The COBOL specific reengineering tools have built-in knowledge of COBOL: They work well if the application at hand conforms to the specific syntax supported by tool, usually the union of several COBOL dialects.

Examples of language-independent tools are Rigi [15] or Ciao [6].

Many papers report on tools and techniques for analyzing C code. We found it difficult to transfer these to the COBOL domain and to apply them to our case studies. COBOL lacks many C features, such as types, functions, and parameters for procedures. Moreover, approaches developed for C tend not to take advantage of typical COBOL issues, such as the database usage for business applications.

Finding reusable modules Part of our work is similar in aims to the RE^2 project, in which candidature criteria have been defined to search for functional abstractions, data abstractions, and control abstractions [7]. The RE^2 approach has been applied to COBOL systems by Burd *et al.* [5, 4].

Neighbors [17] analyzes large Pascal, C, assembly, and Fortran systems consisting of more than a million lines of code. One of his conclusions is that in large systems, module names are not functional descriptions, but "architectural markers". This agrees with our observation that we could not use module names to locate reusable code, while section names proved helpful in many cases.

7 Conclusions

Rapid system understanding, in which fast comprehension is more important than highly accurate or detailed understanding, plays an important role in the planning, feasibility assessment and cost estimating phases of system renovation projects.

System understanding tools require an architecture in which it is easy to exploit a wide range of techniques. The architecture discussed in Section 2 distinguishes analysis of source code, a central relational database to store analysis results, and various forms of presenting these results such as report generation and graph visualization.

The datamodel used by the relational database, and the analysis and visualization techniques used, depend on the rapid system understanding problem at hand. The paper discusses an instantiation for identifying reusable business logic from legacy code.

Lexical analysis, using only superficial knowledge of the language used in the sources to be analyzed, is sufficiently powerful for *rapid* system understanding. An important advantage is its flexibility, making it possible to adapt easily to particularities of the system under consideration. (See, for example, the derivation of the call graph of Share as discussed in Section 4).

In order to assess the validity of the architecture proposed, the emphasis on lexical analysis, and the instantiation used for identifying business logic from legacy code, we studied two COBOL case studies from the banking area. The two case studies show that (1) lexical methods are well-suited to extract the desired data from legacy code; (2) the presentation forms chosen help us to quickly identify business data fields and chunks of code manipulating these fields; (3) the proposed approach is capable of finding answers to the reusability questions posed in Section 3.

We consider the results of this case study to be encouraging, and believe the approach to be viable for a range of system understanding and reusability assessment problems. The limitations of our approach are:

- Lexical analysis cannot take advantage of the syntactic structure of the sources. In our cases, for example, it is difficult to extract those variables that are used in, say, conditions of if-then-else statements.
- Identification of business data is based on the assumption that this data is stored in databases.
- Identification of data field usage is based on textual searches for the field names. This works on the assumption of systematic variable naming. A more accurate, yet also much more involved, method would be to follow the database field usage through the dataflow.

The latter two assumptions are reasonable and will generally hold, but certainly not for all systems.

8 Future Work

While developing the rapid system understanding tools and techniques, and while applying them, several further research questions emerged. We are in the process of investigating the following topics.

Use of metrics Our work bears a close relationship with the area of metrics. A question of interest is what metrics are indicative for reusability in the two COBOL systems we studied. Another relevant question is which metrics can be computed sufficiently easily, in order to make them applicable in a rapid system understanding setting.

Code cloning While analyzing Mortgage we observed a high degree of duplicated code. We intend to investigate whether lexical methods are suitable for detecting the clones present in Mortgage.

Restructuring and modularization We are currently experimenting with applying cluster analysis methods to remodularization of legacy code [8].

Comprehension models Searching through code using lexical analysis can be viewed as browsing in order to answer questions and verify hypotheses. Recent studies in system and program understanding have identified code cognition models emphasizing this hypothesis verification aspect [3, 12, 13]. From our experience with the two COBOL cases we observed that many of our actions were aimed at *reducing the search space*. Thus, rather than verifying hypothesis immediately, we started by organizing the set of programs such that the chance of looking at less relevant programs was minimized. It seems interesting to study how this search space reduction fits in some of the existing code cognition models.

As a last remark, the present year 2000 crisis may be an ideal opportunity to experimentally verify the validity of cognition models for rapid system understanding. Many "year 2000 solution providers" start by performing a "quick scan" in order to determine the costs of the year 2000 conversion project, and almost all of these scans are based on lexical analysis. A successful cognition model should be able to describe most of the methods used by these solution providers, and might be able provide hints for improvements for methods not taking advantage of this model.

References

[1] A. V. Aho, B. W. Kernighan, and P. J. Weinberger. *The AWK Programming Language*. Addison-Wesley, 1988.

[2] M. G. J. van den Brand, A. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. In *Working Conference on Reverse Engineering; WCRE97*, pages 144–155. IEEE Computer Society, 1997.

[3] R. Brooks. Towards a theory of the comprehension of computer programs. *Int. Journal of Man-Machine Studies*, 18:543–554, 1983.

[4] E. Burd and M. Munro. Enriching program comprehension for software reuse. In *International Workshop on Program Comprehension; IWCP'97*. IEEE Computer Society, 1997.

[5] E. Burd, M. Munro, and C. Wezeman. Extracting reusable modules from legacy code: Considering the issues of module granularity. In *3rd Working Conference on Reverse Engineering; WCRE'96*, pages 189–196. IEEE Computer Society, 1996.

[6] Y.-F. Chen, G. S. Fowler, E. Koutsofios, and R. S. Wallach. Ciao: A graphical navigator for software and document repositories. In *International Conference on Software Maintenance; ICSM 95*, pages 66–75. IEEE Computer Society, 1995.

[7] A. Cimitile and G. Visaggio. Software salvaging and the call dominance tree. *Journal of Systems Software*, 28:117–127, 1995.

[8] A. van Deursen and T. Kuipers. Finding classes in legacy code using cluster analysis. In S. Demeyer and H. Gall, editors, *Proceedings of the ESEC/FSE'97 Workshop on Object-Oriented Reengineering*. Report TUV-1841-97-10, Technical University of Vienna, 1997.

[9] E. R. Gansner, E. Koutsofios, S. North, and K.-P. Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, 1993.

[10] W. G. Griswold, D. C. Atkinson, and C. McCurdy. Fast, flexible syntactic pattern matching and processing. In *Fourth Workshop on Program Comprehension; IWPC'96*. IEEE Computer Society, 1996.

[11] L. Markosian, P. Newcomb, R. Brand, S. Burson, and T. Kitzmiller. Using an enabling technology to reengineer legacy systems. *Communications of the ACM*, 37(5):58–70, 1994. Special issue on reverse engineering.

[12] A. von Mayrhauser and A. M. Vans. Identification of dynamic comprehension processes during large scale maintenance. *IEEE Transactions on Software Engineering*, 22(6):424–438, 1996.

[13] A. von Mayrhauser and A. M. Vans. Hypothesis-driven understanding processes during corrective maintenance of large scale software. In *International Conference on Software Maintenance; ICSM'97*, pages 12–20. IEEE Computer Society, 1997.

[14] Micro Focus Revolve user guide. Burl Software Laboratories Inc., USA, 1996.

[15] H. A. Müller, M. A. Orgun, S. R. Tilley, and J. S. Uhl. A reverse engineering approach to subsystem structure identification. *Journal of Software Maintenance*, 5(4):181–204, Dec. 1993.

[16] G. C. Murphy and D. Notkin. Lightweight lexical source model extraction. *ACM Transactions on Software Engineering Methodology*, 5(3):262–292, 1996.

[17] J. M. Neighbors. Finding reusable software components in large systems. In *3rd Working Conference on Reverse Engineering; WCRE'96*, pages 2–10. IEEE Computer Society, 1996.

[18] L. Wall and R. L. Schwarz. *Programming Perl*. O'Reilly & Associates, Inc., 1991.