An Assistant for Re-Engineering Legacy Systems

Zheng-Yang Liu & Mike Ballantyne

EDS Research, Austin Laboratory
1601 Rio Grande, Ste 500
Austin, Texas 78701
liu@austin.eds.com

Lee Seward

EDS CORN Account, A5S-D49
13600 EDS Drive
Herndon, VA 22071

Abstract

This paper presents a knowledge-based approach to evolving and re-engineering large legacy systems. It describes an AI application to one special area in software maintenance—upgrading existing software to new technology standards. This approach assists the human engineers in the conversion of software systems by automating the tedious and knowledge-intensive conversion process. The research shows that the payoff for using the knowledge-based approach to software evolution is not only in terms of time and money saved, but also in terms of improved quality of the upgraded software system.

Introduction

Business software systems, such as bank finance systems and airline reservation systems, are loaded with diverse business policies and corporate decisions and constitute one of the most important industrial assets of any company. Many such systems were developed using the technology of the 1970s to mid-1980s. They have been modified many times by different programmers. As a result, these "legacy" systems have become very complex and difficult to maintain, especially when we must perform necessary and timely technology upgrades for them. The greater the differences between the technology standards, the more difficult and convoluted the task.

The rationale for re-engineering existing, operational business systems instead of developing new software from scratch is based not only on economic but also pragmatic considerations. For these legacy systems, requirements and specifications are extremely difficult to elicit. Often, no single person in a corporation could completely describe what was in the system [Curtis, et al, 1988]. New software is too expensive to develop. Corporations simply cannot afford to wait for, or risk their businesses by using some new software that would operate their business differently. The fate of the enormous collection of existing, operational legacy systems is clear—business software must evolve.

This paper describes our approach to re-engineering and converting a large body of legacy software applications with technology upgrades using AI techniques. The project was COmputer Resource Nucleus (CORN), which EDS contracted to perform in March, 1992 for the Federal Aviation Administration (FAA). The goal of the CORN project, one of the largest outsourcing contracts ever entered into by the Federal government, was to consolidate the platform. Some of the FAA's applications, consisting of over 2 million lines of Data General (DG) AOS/VS Interactive COBOL code, were running on DG minicomputers placed at various locations throughout the United States. The targeted platform chosen by FAA was an IBM ESA teleprocessing environment, using CICS, TSO, COM-PLETE and INTERCOMM TP monitors. The conversion task involved redesigning the applications' user interface and database transactions, as well as converting the language and environment features so that the applications could support numerous remote terminals and provide faster access to application data. Over 1,500 DG AOS/VS Interactive COBOL programs had to be converted to IBM CICS COBOL II and IBM batch COBOL II for on-line, batch, and split on-line/batch applications. Typically, there was no documentation of design and no maintenance history available for these legacy systems. The source code provided the sole definition of the systems.

The processes involved in the conversion task did not lend themselves well to automation using traditional techniques due to the complexity and sheer volume of the code and the requisite knowledge of both source and target environments. An initial experiment using plan recognition and planning for program understanding and conversion did not succeed because the scale and complexity of the problem renders these approaches intractable. We have built a knowledge-based system, COGEN (COde GEneration), which consists of a central knowledge base of expert conversion rules and a set of tailored CASE tools for the conversion task. Fundamental to this approach is the idea of treating a program as an artifact with structure, behavior and function. The conversion thus involves restructuring the original program constructs to produce expected behaviors for the overall function of the upgraded system. COGEN works as an intelligent assistant to our system engineers (SEs) by automating the tedious and knowledge-intensive conversion process.

COGEN was deployed in January 1993 and has since been heavily used daily by the EDS SEs of the CORN conversion teams located worldwide, saving tremendous amounts of time and resources which would have had to be expended without the knowledge-based tools. This work demonstrated that the payoff for using the knowledge-based approach is not only in terms of time and money

saved, but also in terms of improved quality of the upgraded system.

Problem Description

We were given three overall constraints by FAA for the CORN conversion task. First, the converted system had to be functionally equivalent to the original system. Second, there should not be any additional complexity for the user interface. Third, we should not emulate any ability of the original system that is not directly available in the target environment. The conversions should be finished within two years.

To show the complexity of the conversion problem, we first describe the crucial differences between the two technological standards.

Restructuring User Interface

On a minicomputer with the DG COBOL environment, the user-computer interface for on-line programs is based on field-by-field interactions. The computer processes each field of data entered by the user promptly and then awaits the next input by constantly polling the user's terminal device for data from the user. This mode of transaction is referred to as the conversational style. In contrast, the new user interface on a CICS teleprocessing platform is based on screen-by-screen interactions (Suhy, 1991). Whenever a screen image is sent to a terminal, the program stores any data it has processed up to that point, frees the resources used, and ends temporarily. When the user enters more data and enters an attention identifier key (function keys, attention keys, or ENTER), the program is restarted along with the stored data, and processing continues until the next screen image is sent out by the program. This mode of transaction is referred to as the pseudo-conversational style. It enables CICS to support numerous remote user terminals because resources are not tied up while the user is entering data, prior to pressing an attention identifier key.

Re-Writing Database Transactions

Both on-line and batch programs in the original legacy systems used file processing for defining, searching, reading, and writing records with a local DBMS or other type of file structure. Typical file and record operations include READ, WRITE, REWRITE, START, FIND, CONNECT, RECONNECT, GET, OBTAIN, ROLLBACK, INITIATE, READY, STORE, ERASE, MODIFY, etc. In contrast, the target environment uses ADABAS SQL, VSAM, or sequential file access commands. Original file and record definitions and the file access commands must be redefined to use the ADABAS SQL, VSAM, or sequential file commands. Moreover, during conversion additional data structures must be created—for example, ADABAS aliases and cursors—and new procedures built and integrated in to save and restore the data between the pseudo-conversational CICS calls. To do this correctly, information must be collected and analyzed for each instance of a file access command, regarding how a particular record is defined, where it is accessed, and how the new environment restricts the type of operation.

Translating Language Features

Additionally, from the DG COBOL environment to the IBM teleprocessing platform, there are many language "dialect" differences and environmental differences which must be converted. These changes concern all aspects of an application program. These changes are generally not hard to understand for those well-versed in the two environments but are formidably tedious, time-consuming and error-prone, and warrant the development of automated tools for converting them.

In preliminary test conversions with a program which has over 7,000 lines, it took an experienced SE two weeks to manually translate a single feature—converting DG DBMS FIND statements to equivalent ADABAS SQL commands. The SE was looking for patterns of a certain construct and rewrote them as required for the target environment. The tools available then were a text editor and some simple operating system utilities, which are inadequate in terms of efficiency, flexibility and providing guidance for conversion.

Most conversion instances were not one-to-one, but depended on analysis of the contextual information that throws light on their meanings and thorough knowledge of the two platforms. The vendor market does not have any suitable tools that are directly available for the task. Much of the research in software engineering has concentrated on methods for developing new software rather than methods for analyzing, converting and evolving existing software [Ward & Bennett, 1993], even though studies [Schindler, 1990, O'Hare & Troan, 1994] indicate that more than half the funds expended for software in the United States are for software maintenance. For the CORN project, estimates were that the conversion project could not be finished in time if only manual conversion was pursued. Worse yet, the system could become even more difficult to maintain in the future due to the inconsistent translation styles that would likely be introduced by different conversion SEs.

As the time pressure, problem complexity, and requisite knowledge involved in this task overwhelm humans, we were forced to address the challenging question: how can we develop adequate knowledge-based tools to assist the SEs to technology-upgrade large legacy systems?

Design and Implementation

In response, our approach to developing knowledge-based tools for converting large legacy systems centers around capturing and modeling the expert knowledge of the SEs in terms of conversion rules. Clearly, an SE capable of doing the conversion must be knowledgeable in both the technology standards. We observed that much of what goes on in the conversion activities consists of judgmental processes based on essential mapping from the source platform to the target environment and on following certain

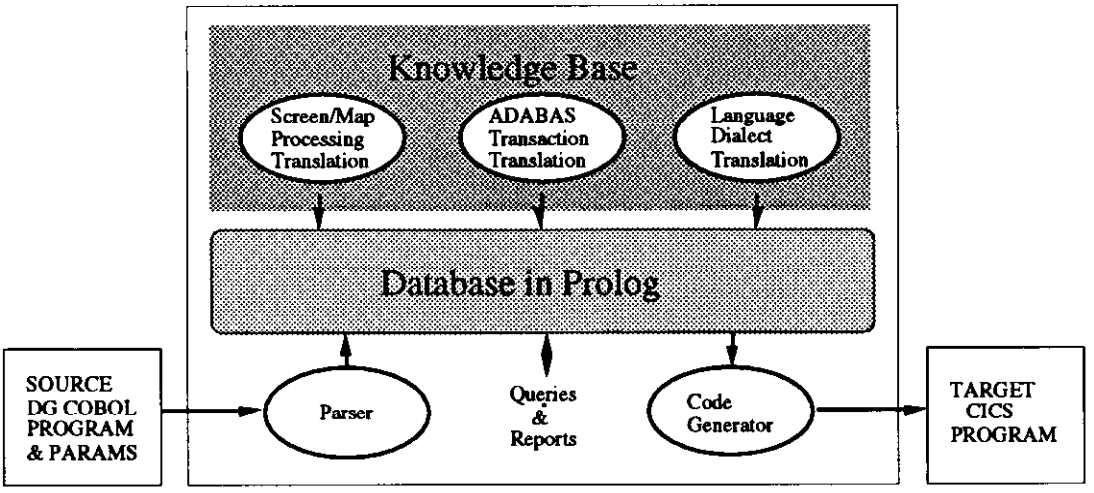

Figure 1: COGEN Architecture

guidelines. The mapping and guidelines naturally fall into the category of expert system rules.

System Architecture

The COGEN knowledge base and the translator were implemented in Quintus Prolog, with supporting tools like Perl, UNIX scripts and C (Lex/Yacc) on SUN/SPARC workstations. An identical PC version of COGEN is also available for remote-site SEs. In designing and developing COGEN, conventional software engineering techniques (Liu & Yeh, 1984) were integrated with the knowledge-based approach to provide a set of tools for the conversion task. Figure 1 shows the general architecture of COGEN.

Briefly, COGEN includes a special parser which loads a source program into a deductive relational database in the form of a program syntax tree. The data definitions are captured in a symbol table. Queries can be entered into the database to obtain various kinds of useful information about the program's structure and behavior in terms of data and control flow analysis. To convert the program, the translation rules are applied to restructuring the program in the database, creating new facts describing the program in the new environment and altering the original syntax tree with new statements added and old statements commented out. Advising messages are also generated in certain cases for the SEs. Finally, the code generator traverses the new syntax tree of the program and produces the result of the translation in the form of a new program for the SEs. The net result is that the user-interface code gets restructured, the file processing operations redefined, and the language dialect features converted.

The SEs can guide COGEN at various stages in the translator by preparing and fine-tuning a set of parameters to the translator. The parameters indicate which category of on-line or batch the program belongs to, the new program name for replacing the old name, whether or not to suppress the translator's advising or warning messages during translation, and so on. The SEs can also disable or enable any translation rules by setting certain parameters.

Knowledge Base

In acquiring and representing the conversion rules, we view each program as an artifact, like a physical device (Bobrow, 1985; Liu & Farley, 1990; Liu, 1993), with structure and behavior and satisfying a certain business function. The term structure refers to the data entities, relations among them, procedures to manipulate them, and how they are organized as a program. The term behavior refers to the time course of observable changes of state of the entities and data/control flow of the program. Each program in the legacy system is perceived to have a function: a relation between what the program is designed for and its actual behavior.

With this view, two types of reasoning become applicable: causal and teleological. By causal reasoning, we mean that the behavior of the system is generated by the structure of the system, *i.e.*, structure causes behavior. For teleological reasoning, we mean that the function constrains the behavior of the system, *i.e.*, behavior must satisfy function. To understand why a particular program component behaves in a certain way, causal reasoning would point to the structural aspects of the program, while teleological reasoning would point to the purpose it tries to satisfy for the program.

For instance, converting field-based interactions (DG) to screen-oriented interaction (CICS) meant that one had to partition the DG code into logical chunks that effectively handled one screen and within each chunk one had to determine which fields should be grouped together and processed as a whole. This determination could not be made from the code alone. One had to reason about the role that screens play in the overall user interface.

The concepts of structure, behavior, and function are inseparable and crucial for re-engineering an artifact.

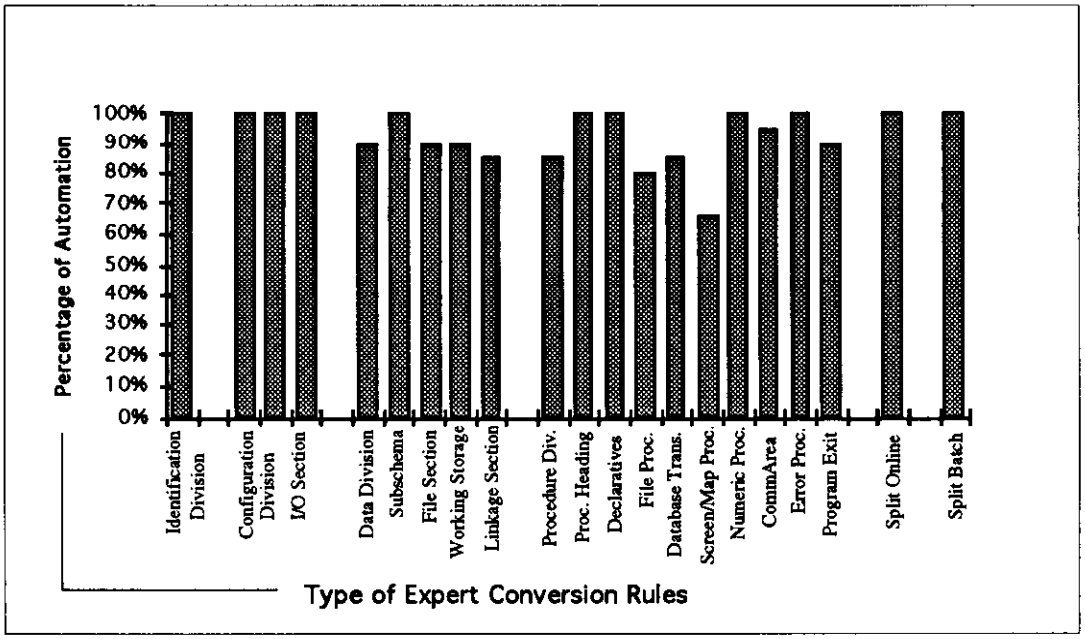

Figure 2: Percentage of Conversion Automated by COGEN

During conversion, when we change a program construct in an old program, we expect it to behave in a way which satisfies the program's overall function in the target environment. The function constrains the behavior of the program which, regarding the target environment, ultimately points to the structural component of the original program to be changed. Reasoning about function also allows interesting optimizations in original programs, and mapping rules can be established. Completely different structure may be substituted for a construct in a program if the two provide the equivalent functions.

To meet the FAA's requirement that the converted system must be functionally equivalent to the original system, any restructuring of the old programs must integrate smoothly with the backbone of the original code. To enable it to work in the target environment, which expects a different set of behaviors, we focused on searching for the structural units that pertain to the expected behavior changes in the target environment.

To acquire the knowledge for conversion we worked interactively with the SEs who were knowledgeable in the two technology standards. We provided them with tools to automate their suggestions to us of rule additions and revisions in the early stage of COGEN research. As CICS is on-line, we were also careful about the performance factor of the converted programs. A total of 246 expert rules were identified, created and implemented in COGEN. Figure 2 shows the distribution of the rules for restructuring different parts of a COBOL program and an estimation of automated conversion achieved by applying each category of the rules to each part.

It was difficult at first to extract conversion rules out of the SEs, as some conversion rules are very complex and involved. In different situations, one program statement can be translated in very different ways. For example, a DG DBMS FIND statement can be converted to different ADABAS SQL commands depending on the value of FIND's direction attribute as well as the context in which the FIND statement is used. We identified each of the situations as different cases and explicitly created a different rule for each case. This "divide-and-conquer" approach enabled us to express all the conversion rules we could identify. For example, there are seven conversion rules for translating a DG DBMS FIND statement in different situations. The SEs found the process of rules identification rewarding as they saw the result of rules applied during conversion. They often took the initiative to suggest new rules to be included in the knowledge base.

A conversion rule has certain preconditions which must be true in the deductive database in order for the rule to be fired. The choice of representing program statements in the database makes such inference and cross reference easy and straightforward. During conversion, a rule can be applied repeatedly many times as needed using the built-in backtracking mechanism in Prolog.

Each type of statement has its database schema. For example, the database schema of a DG DBMS FIND statement is:

```
stmt(Id,FIND(Direction,Member,Rec_cursor,
    Within,Criteria,Criteria_field,Assign,
    At_end_statements,At_error_statements)).
```

In the situation where two FIND statements are involved in an IF construct as

```
IF BLD-SWITCH = 'N'
    FIND FIRST BLD WITHIN REGION-BLD
ELSE
    FIND NEXT BLD WITHIN REGION-BLD
```

```
EXEC ADABAS
     FIND
     DECLARE Fxxx CURSOR FOR
     SELECT find:Criteria_Field, find:Criteria_Field
        ... if find:Criteria_Field not NULL |
             '*' if find:Criteria_Field is NULL}
     FROM faa_dbms_name:ibm_name using find:Rec_cursor as
           faa_dbms_name:dg_name Axxx
     WHERE faa_dbms_name:list_of_keys = :<value>
     OPTIONS PREFIX = Axxx-
END-EXEC
   insert message DBA008 here
EXEC ADABAS
   OPEN Fxxx
END-EXEC
EXEC ADABAS
   FETCH Fxxx
END-EXEC
IF ADACODE = 0
   MOVE Axxx-find:Criteria_Field TO find:Criteria_Field
   MOVE Axxx-find:Criteria_Field TO find:Criteria_Field
        if find:Criteria_Field not NULL |
   MOVE Axxx TO faa_dbms_name:dg_name using find:Rec_cursor as
       faa_dbms_name:dg_name  if find:Criteria_Field is NULL}
   insert message DBA023 here
ELSE
   insert message DBA009 here
EXEC ADABAS
     CLOSE Fxxx
END-EXEC
     place find:At_end_statements here, if any
END-IF
insert message DBA006 DBA026 here
```

* Fxxx and Axxx above are the cursor name and related alias name to be created for SQL commands; *faa_dbms_name* is one of the parameters supplied by a conversion SE.

Figure 3: A Rule Template in COGEN's Knowledge Base

the construct is converted to its equivalent ADABAS SQL commands based on the translation rule which is shown in Figure 3.

In the output generated by COGEN, the statements which are supposed to be deleted from the original program are commented out with the special label 'FAACOM*'. Newly inserted statements are marked with labels like 'FAADBA' or 'FAAINS'. This allows the SEs to easily compare and verify the result of the translation. The statements that are not changed are printed out verbatim.

For example, the output generated by COGEN after converting the above FIND statements is displayed in Figure 4.

Development and Deployment

A characteristic of the COGEN knowledge base is that it is open-ended. Due to the time pressure at the early stages of COGEN project, it was deployed and used before it was "completed." We adopted a phased-in approach to releaseing COGEN. This resulted in an automated email protocol that allowed the SEs to use the translator through email.

The first version of COGEN was released on January 26, 1993. Instead of giving runtime versions of the translator to remote-site SEs in the early stages, we let the SEs use COGEN via email transactions following a pre-defined protocol. When such a mail message arrived, its subject line and the message body would determine what service was requested. COGEN performed the service and then sent back the result of the translation to whomever had made the request. This allowed our R&D laboratory to deliver solutions on a variety of heterogeneous computing platforms and yet continued to allow development of the underlying system to add more rules to the knowledge base. This kept us from straying too far from the

```
FAACOM* IF BLD-SWITCH = 'N'
FAACOM*  FIND FIRST BLD WITHIN REGION-BLD
FAACOM* ELSE
FAACOM*  FIND NEXT BLD WITHIN REGION-BLD.
FAACOM* IF HIT-SW = 'N'GO TO 109-EXIT.
FAADBA  IF BLD-SWITCH = 'N'
FAADBA     EXEC ADABAS
FAADBA        FIND
FAADBA        DECLARE F001 CURSOR FOR
FAADBA        SELECT *
FAADBA         FROM AR-BLD A001
FAADBA         WHERE REG-BLD-KEY = :<value>
FAADBA         OPTIONS PREFIX = A001-
FAADBA     END-EXEC
FAAINS*WARNING (OL_PD_DB_5: SEE NOTE DBA008)
FAADBA       EXEC ADABAS
FAADBA            OPEN F001
FAADBA       END-EXEC
FAADBA       EXEC ADABAS
FAADBA            FETCH F001
FAADBA       END-EXEC
FAADBA  ELSE
FAADBA       EXEC ADABAS
FAADBA            FETCH F001
FAADBA       END-EXEC
FAADBA  END-IF
FAADBA  IF ADACODE = 0
FAADBA      MOVE A001 TO BLD
FAAINS*WARNING (OL_PD_DB_5: SEE NOTE DBA023)
FAADBA        ELSE
FAAINS*WARNING (OL_PD_DB_5: SEE NOTE DBA009)
FAADBA       EXEC ADABAS
FAADBA            CLOSE F001
FAADBA       END-EXEC
FAADBA       GO TO 109-EXIT
FAADBA  END-IF
FAAINS*WARNING (OL_PD_DB_5: SEE NOTE DBA026)
FAAINS*WARNING (OL_PD_DB_5: SEE NOTE DBA006)
```

Figure 4: COGEN Services through Email

knowledge engineering tasks and allowed us to focus on developing the conversion rules.

We let all the SEs involved in this project know that COGEN was designed as an assistant to the SEs doing the conversion, and helped them realize that quality is a responsibility and its continuous improvement not a cost but a savings. To allow them to contribute their ideas, the email protocol included special subjects for suggesting new rules, reporting bugs, requesting changes, and making comments. All suggestions about rule additions and revisions went through a contact SE who coordinated and formally requested the rule changes. These requests for adding new rules reflected the collective wisdom of the SEs in the field and greatly improved the translator's conversion capability. Figure 5 shows a list of major COGEN services available through the email protocols.

```
Registering/Updating SE Information
Retrieving a Directory Listing by WBS
Updating and Parsing Source Code
Converting a Program
Retrieving Source Code
Renaming a Source Code File
Deleting Source Code
Updating System Parameter File
Retrieving a System Parameter File
Reporting a Bug
Offering a Suggestion of New Rules
Requesting a Rule Revision
Requesting a Summary Report
Suggesting an Issue for Discussion
Requesting a Translation Log
```

Figure 5: COGEN Services through Email

In addition, the email protocol system performed periodic management that recorded, reported and transformed information between remote users and the translator. This allowed us to monitor the use of the translator on-line and automatically routed error messages to those who needed them so that these errors could be corrected quickly. Later, we added runtime versions of COGEN that run in the UNIX and MS-DOS environments.

Use and Payoff

The conversion SEs using COGEN should be knowledgeable with regard to the source and target platforms so that they are able to verify the translation. To use COGEN, an SE can convert one program at a time. The SE can also use COGEN to convert a group of programs in a batch. The SEs welcome this because the batch can run during a lunch break or at night on an unattended machine. In this conversion mode, COGEN generates and saves any advising messages regarding each program conversion in a report, which the SE can read later. The SE can then take appropriate action.

The conversion of an on-line program with screen processing is the most difficult part and proceeds in three distinct stages. First, with some help from a conversion SE, COGEN partitions the DISPLAY/ACCEPT statements in the program into CICS screens and organizes the program into sections responsible for processing those screens. The SE usually inserts special tokens in the program before conversion to indicate heuristically where possible maps begin and end. COGEN generates the CICS physical maps, a special type of the assembler mapset program, from the screen section statements and the user-inserted tokens.

Second, the additional conversion rules translate the resulting program into a conversational CICS program. As a result of the translation, the map images, data linkage section with saves and restores for passing data to and from the CICS monitor, and file access commands are created based on the analysis of the entire program. The original constructs that previously served equivalent purposes are deleted (commented out). As the field-by-field user interface is intertwined with the complex program logic, the original code for the user interface is torn apart and completely rewritten. The SEs usually must correct the output generated by COGEN, as the screen building heuristics are incomplete. The SEs first assemble the mapset and resolve any conflicts in the maps by adding or deleting fields or altering their attributes. They may re-tokenize the program to let COGEN convert it again. Like running a simulation, the SEs can tailor the output of the translator by adjusting the program conversion parameters or inserting tokens in different ways.

Finally, COGEN reparses this corrected program and algorithmically restructures it into a pseudo-conversational CICS COBOL program. A process of "mental simulation" is performed as the SEs trace through the program logic to verify the results of the translation. The resulting program now is ready for the unit test.

For converting batch programs, no screen processing is involved, and the translation is relatively straightforward. COGEN has been very successful in restructuring batch programs and in converting local DBMS commands to ADABAS SQL constructs and other DG-to-IBM-CICS feature translations automatically. When COGEN cannot reach a detailed solution, it still assists the SEs by printing out a skeleton that they can then fill in.

The effort to design and implement the knowledge base took about 3 man years. The payoff in exact dollar figures is hard to quantify at this point. Nonetheless, it is clear that using this AI solution to convert legacy systems saves time and resources, and offers definite benefits, the greatest being the ability to convert all the programs in a legacy system uniformly.

Due to the complexity of restructuring a conversational DG program into a pseudo-conversational CICS program, the translator's performance varies, yet it still performs remarkably. It is estimated that an average of 65% of the code changes is automated for screen and map processing with on-line programs. For translating language features and general batch programs, the automation of conversion is nearly 100%. The additional payoff includes increased maintainability of the systems, reuse of the methodology for similar future applications and, most importantly from a business standpoint, completion of the project on time.

Maintenance

The COGEN translator was designed to be able to incrementally increase its knowledge base with a minimum amount of maintenance. RCS, Revision Control System, (Tichy, 1991), is used for automated version control of all the source codes of COGEN. Currently, one research scientist, who is one of the original designers of the system, is supporting the system. As the system is transparent to the users, good suggestions from the SEs can be absorbed incrementally due to its rule-based nature and modular design. Requests for changes in the translator were frequent in the early stages of developing the system; sometimes in one day there were several releases of the translator (parser, converter, and automated e-mail protocol subsystem) as a result of the rule revisions, bug fixing, and requirement changes. COGEN has stabilized with the heavy use of the translator since it was first released.

When we make any changes to the translator and make new releases, we again take advantage of electronic mail to send relevant pieces of code to the remote-site SEs. We compress the code and send it over the net, and when the SEs receive the mail, all they need to do is uncompress it before they use the new version of the translator. Lately, it has usually taken a couple of hours to release a new version of COGEN after an SE's request for change or bug fix in the translator is sent to us.

With both the DG and CICS platforms, the knowledge in this domain is not likely to change. To date, the translator has been an integral part of the CORN conversion plan, offering assistance to the SEs by automating tedious and error-prone manual conversion. The translation rules are stable and readily used.

Discussion

The importance of evolving and re-engineering existing, operational legacy systems is a point of consensus among all colleagues who have studied the issue of software reverse engineering, whether in design recovery (Chikofsky & Cross, 1990; Jain & Williams, 1993), in structured analysis of source code (Hartman, 1990; O'Hare & Troan, 1994), in user interface reverse engineering based on structural and behavioral representations (Merlo, et al., 1993), or in extracting domain models for reuse (Devanbu & Frakes, 1992). COGEN converts existing, operational systems so that they work in a new environment. It is a function-preserving transformation. We strive for pragmatic solutions commensurate with the needs of a given conversion task.

Looking back at what we have done, we have learned many important lessons in managing a project of this size. First and foremost, it is critical that the solutions be task-driven, customer-oriented. We keep the users in the loop in the whole development process, ready to adjust and tailor the solutions to their needs.

Developing an expert system is a special form of engineering. To date, many AI techniques and ideas are mature enough to be applied to real-world problems and deliver solutions. Fundamental to the success of building an expert system for real-world applications is the process of knowledge acquisition and representation. In COGEN, we facilitate the SEs' contribution of their expertise and encourage them to suggest and formulate conversion rules to be put into the knowledge base. It makes the solutions easy to understand and acceptable. The rules are organized

in such a way that the translator slows down linearly as more rules are added.

One encouraging thing we learned through the COGEN research is that COGEN's framework can be adapted to similar applications for converting other legacy systems. It becomes clear that loading a legacy system into a relational database provides a uniform representation for different program constructs and offers the opportunity for automated reasoning about the system and revising its structure or generating new constructs in different target languages. As conversion is always task-specific, if we need to convert COBOL programs into C or C++ programs, for example, we must create and load COGEN with a new set of conversion rules tailored for the task.

Conclusion

With the ever-increasing advances of new technologies, technology upgrades for large software systems are becoming more and more frequent and costly. This corresponds to the well-known fact that software maintenance activities account for the major cost in the software life cycle. With this trend, we believe that timely technology upgrades of large legacy systems are only possible with carefully engineered knowledge-based CASE tools to help rewrite the original software. Re-engineering in general is very hard (Neches, et al., 1985) as it is difficult to capture and model the conversion expertise of the SEs and tailor it for various conversion activities. Based on the perception of viewing software with structure, behavior and function, most program conversion knowledge can be explicitly represented in an automated conversion system. This approach used in COGEN made possible what was once thought to be impossible with traditional approaches.

Acknowledgments. We would like to thank Jim Johnson, Neil Iscoe, Bill Buratti, Julie Francis, Jeff Jacobs, and Betty Milstead for their help in this work.

References

Bobrow, D. G., Ed. (1985). *Qualitative Reasoning about Physical Systems*. Cambridge, Massachusetts, MIT Press.

Chikofsky, E. J. & Cross, J. H. (1990). Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software* **7**(1): pp. 13-17, January.

Curtis, B., Krasner, H. & Iscoe, N. (1988). A Field Study of the Software Design Process for Large Systems. *Communications of the ACM*, **31**(11): pp. 1268-1287, November.

Devanbu, P. & Frakes, W. (1992). Extracting Formal Domain Models from Existing Code for Generative Reuse. In *Proceedings of AI and Automated Program Understanding Workshop*, San Jose, CA, July.

Hartman, J. E. (1990). *Automatic Control Understanding for Natural Programs.* Ph.D. Dissertation, Department of Computer Sciences, University of Texas at Austin.

Jain, P. & Williams, G. (1993). Business Process Redesign Support: Modeling, Analyzing, and Evolving Large Scale Business Processes. In *Proceedings of AAAI-93 Modeling in the Large Workshop*, Washington, D.C., July.

Liu, Z.-Y. & Yeh, R. M. (1984). *Database Support for Software Development*. Report No. 84-2074, Computer Science, University of Maryland, College Park.

Liu, Z.-Y. & Farley, A. M. (1990). Shifting Ontological Perspectives in Reasoning about Physical Systems. In *Proceedings of AAAI-90* (pp. 395-400). Boston, MA.

Liu, Z.-Y. (1993). Qualitative Reasoning about Physical Systems with Multiple Perspectives. *AI Magazine*, **14**(1): pp. 77-79, Spring.

Merlo, E., Girard, J. F., Kontogiannis, K., Panangaden, P. & DeMori, R. (1993). *Reverse Engineering of User Interfaces.* Technical Report, McGill University and Centre de recherche informatique, Montreal, Canada.

Neches, R., Swartout, W. R. & Moore, J. D. (1985). Enhanced Maintenance and Explanation of Expert Systems through Explicit Models of their Development. *IEEE Transactions on Software Engineering*, **SE-11**(11): pp. 1337-1351.

O'Hare, A. B. & Troan, E. W. (1994). RE-Analyzer: From Source Code to Structured Analysis. *IBM Systems Journal*, **33**(1): pp. 110-130.

Schindler, M. (1990). *Computer-Aided Software Design.* New York, John Wiley & Sons.

Ward, M. P. & Bennett, K. H. (1993). A Practical Program Transformation System for Reverse Engineering. Technical Report, Computer Science Department, Science Labs, Durham University.

Waters, R. C. (1985). The Programmer's Apprentice: A session with KBEmacs. *IEEE Transactions on Software Engineering*, p.1296, November.

Schindler, M. (1990). *Computer-Aided Software Design: Build Quality Software with CASE*. New York, John Wiley & Sons.

Suhy, A. M. (1991). *CICS Using COBOL*. Belmont, CA, Wadsworth Publishing Company.

Tichy, W. F. (1991). RCS—A System for Version Control Department of Computer Sciences, Purdue University. West Lafayette, Indiana 47907.