Software Reverse Engineering: A Case Study

ERIC J . BYRNE

Department of Computing and Information Science, Kansas State University, Nichols Hall, Manhattan, Kansas 66506, U.S.A.

SUMMARY

This paper presents lessons learned from an experiment to reverse engineer a program. A reverse engineering process was used as part of a project to develop an Ada implementation of a Fortran program and upgrade the existing documentation. To accomplish this, design information was extracted from the Fortran source code and entered into a software development environment. The extracted design information was used to implement a new version of the program written in Ada. This experiment revealed issues about recovering design information, such as, separating design details from implementation details, dealing with incomplete or erroneous information, traceability of information between implementation and recovered design, and re-engineering. The reverse engineering process used to recover the design, and the experience gained during the study are reported.

KEY WORDS Design recovery Reverse engineering Structured design Re-implementation Re-engineering Language translation

INTRODUCTION

For the problem of reimplementing an old Fortran program in Ada and providing new documentation, a typical solution is to translate the Fortran source code directly into Ada and document the new program. This paper describes the author's experience with an alternative solution. This alternative applies software reverse engineering techniques to Fortran source code. Design information is recovered from the source code and any existing documentation. This recovered design information is then used to implement a new version of the program written in Ada and to generate up-to-date documentation. The process of recovering a program's design is called design recovery and is an element of software reverse engineering.

Software reverse engineering ¹ is defined as the process of analysing a system to identify the system's components and their interrelationships, and to create representations of the system in another form or at a higher level of abstraction. There are many types of program information that can be abstracted and examined . ²⁻⁵ With an older software system it is particularly useful to abstract and record its design. A design is recovered by piecing together information from the source code, existing documents, personnel experienced with the system and application domain knowledge. Design recovery differs from reverse engineering by emphasizing the recovery of application domain knowledge that helps achieve informative higher level abstractions beyond those created by examining only the source code. Ted Biggerstaff in his article ⁶ on design recovery stated that:

0038–0644/91/121349–16\$08.00 © 1991 by John Wiley & Sons, Ltd.

Received 22 May 1990 Revised 25 June 1991

The recovered design abstractions must include conventional software engineering representations such as formal specifications, module breakdowns, data abstractions, data-flows, and program description language. ... design recovery must reproduce all the information required for a person to fully understand what a program does, how it does it, why it does it. . . .

Modern software designers and developers have useful techniques and methodologies derived from years of experience and observations to guide them in their work. Automated tools have been collected into environments that support the development of software. These tools embody software engineering techniques that assist with the development of software that is well structured, understandable, easier to maintain, contains fewer errors and is properly documented. By recovering the design of an older system, these modern techniques can be applied to transform the design, and to produce a better implementation. This is software re-engineering.

This paper reports on an experiment with reverse engineering conducted at the U.S. Air Force's Avionics Laboratory, System Avionics Division, Avionics Logistics Branch at Wright-Patterson AFB. A Fortran program was selected that is used in a simulation environment for the real-time testing and analysis of the Fire Control Computer used on F-16 aircraft. This simulation environment simulates the subsystems of an F-16 aircraft in an operational environment, including models for the aircraft's aerodynamics, on-board computers, avionics sensors, weapon systems and targets. The simulation environment is composed of separate programs that execute in a distributed environment. The programs execute concurrently and communicate via shared memory. In the autumn of 1988 a project was started to rewrite the existing Fortran code in Ada. The source code is manually translated using source-to-source techniques. Documentation for the translated programs is produced separately by a different group.

The selected program for this experiment was a software model of the Air Data Computer (ADC) used in F-16 aircraft. This is a small program, only several hundred lines of code. The goal of this experiment was to gain experience with software reverse engineering and identify possible problems. A secondary goal was to use a software development environment to record the recovered design information so that the suitability of the environment for supporting reverse engineering could be evaluated. The reverse engineering process began with the existing Fortran source code and documentation for the ADC model, extracted the software design and recorded the recovered design using the software development environment tool. The recovered design was used to generate new documents and to create a new implementation of the ADC model in Ada. Figure 1 shows an abstract representation of the experiment.

WHY USE REVERSE ENGINEERING?

The motivation for this experiment derived from a single problem: given an existing software system written in Fortran, how can an equivalent system, written in Ada, be developed? The problem of reimplementing an existing system in a different programming language has been around for years and three general approaches have emerged: ⁹

1. Manually rewrite the existing system.

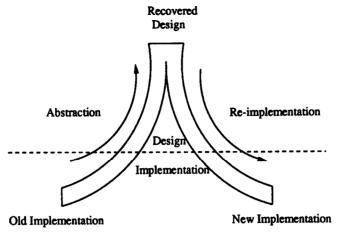


Figure I. Reverse engineering and re-implementation process

- 2. Use an automatic language translator.
- 3. Redesign and reimplement the system.

In the first approach, manually rewriting the existing software system means manually translating from the source language to the target language. With this approach no new software tools are required. There is flexibility in terms of translating the system and changing the system structure. However, there are several disadvantages as well. Manually translated source code often retains the style and flavour of the original implementation. Chances to change the system are often not planned. Individual programmers tend to redo as they work, instead of planning the changes in advance. This approach is time consuming. The number of lines of code that can be translated manually per programmer per day is small. Finally, this is an error-prone approach. Humans make mistakes. There is no means to guarantee that the rewritten system is functionally equivalent to the original. The new system must be thoroughly tested to achieve confidence in it.

The second approach, automatic translation, relies on the use of a tool that accepts software written in the source programming language and generates new source code written in the target language. It is possible for the translation process to be done with little or no human intervention, thereby avoiding many of the problems with the manual rewrite approach. This approach generates new code quickly. However, this approach has several disadvantages. The source language may not yield itself to simple translation into the chosen target language. Some automated translator tools only do the easy part of the translation and leave difficult portions for a human. Automatic translation tends to focus solely on language transformation and does not address the issue of program structure modifications. ¹⁰ The most commonly recognized problem with literal translation is summed up as: garbage in, garbage out. If the existing system is not well-structured, both in terms of its architecture and control-flow, then the resulting system will be of the same poor quality. Automatically generated code may be terribly inefficient. The code produced by translation may also be difficult to understand, greatly increasing its future maintenance costs. ¹¹

Thus, automatic language conversion may not provide an easy route to a complete conversion to a new system.

The third approach is to redesign and reimplement the system. This approach starts with the requirements for the current system and builds a completely new system in the target language. This new system is required to be functionally equivalent to the original system even though it is not derived from it. Of the three approaches, this approach has the greatest chance of producing the best possible new system. Redesigning and reimplementing the system in the new language provides the most power and the greatest flexibility in terms of creating the end product. The resultant system may have significantly lower maintenance costs than systems generated by the other approaches. Finally, redesigning the existing system for implementation in the target language allows for better use of the features of that language. However, this approach also has several disadvantages. It is more difficult than doing an initial design, because of the requirement to emulate the existing system interfaces. This approach has the highest initial cost. It is equivalent to building a new system. The most serious disadvantage is that for many systems it is not possible to redesign from the system requirements, since the requirements may not exist. For many older systems the only accurate statement of the system's capabilities and functionality is often the source code itself. There often is no valid requirement specification for the system.

Reverse engineering provides a new approach. If there is no requirement specification for a system, reverse engineering the system can produce a reconstructed design that captures the functionality of the system. The design should be represented at an abstraction level that removes implementation language dependencies. This makes it possible to reimplement the system in a new language. In addition, the reconstructed design can be transformed to modernize it, restructure it, incorporate new requirements, etc. This is software re-engineering. Thus re-engineering based on a reverse engineering process offers many of the advantages of the redesign and reimplement approach. This approach is always feasible if the source code for a system exists.

Is this a cost-effective solution? It is commonly stated that tool support is necessary to be able to effectively reverse engineer large systems. One project reverse engineered a 24,000 LOC (lines of code) system to respecif y and redocument the system. With tool support it was possible to statically analyse and document the system in one week. It was estimated that it would have taken three man years to document the system manually at the same level of detail. The generated information was used to respecify the system. Respecification took 17 man-months to complete. This amounted to one man-month of specification per 1400 lines of code. In another project, 100,000 LOC comprising over 60 programs were redesigned by manually reconstructing a design representation of the original source code. In addition, 52, 000 LOC of new software was written. The original system totalled 1.5 million LOC. This project was completed in 21 months.

Considering these approaches with respect to the work at the U.S. Air Force's Avionics Laboratory, the programs forming the F-16 simulation environment are currently translated manually into Ada; Several automatic translators were reviewed, but no suitable translator was found. No requirement specification for the selected program, the Air Data Computer (ADC) model, was available. Thus reverse engineering was selected as the alternative method to explore. The purpose of the

experiment reported in this paper was to examine the use of reverse engineering to generate a language-independent design and to create valid documentation for the program. The goal was to produce a design that could be used to implement an Ada program that was free of Fortran characteristics. This paper describes the reverse engineering process used and the problems encountered. The emphasis is on the application of the reverse engineering technique. The issue of cost effectiveness was not explored.

REVERSE ENGINEERING PROCEDURE

The reverse engineering process begins by extracting detailed design information, and from that extracting a high-level design abstraction. Detailed (low-level) design information is extracted from the source code and existing design documents. This information includes structure charts, data descriptions and PDL to describe processing details. A similar approach, but automated, is described elsewhere to recover Jackson and Warnier/Orr documents from code. ¹⁴ The high-level design representation is extracted from the recovered detailed design and expressed using data-flow and control-flow diagrams. Throughout this paper the term 'recovered design' will be used to denote the extracted design. The procedure steps are discussed below. Figure 2 summarizes the procedure.

- 1. *Collect information*. Collect all possible information about the program. Sources of information include source code, design documents and documentation for system calls and external routines. Personnel experienced with the software should also be identified.
- 2. Examine information. Review the collected information. This step allows the person(s) doing the recovery to become familiar with the system and its components. A plan for dissecting the program and recording the recovered information can be formulated during this stage.
- 3. Extract the structure. Identify the structure of the program and use this to create a set of structure charts. Each node in the structure chart corresponds to a routine called in the program. Thus the chart records the calling hierarchy of the program. For each edge in the chart, the data passed to a node and returned by that node must be recorded.
- 4. Record functionality. For each node in the structure chart, record the processing done in the program routine corresponding to that node. A PDL can be used to express the functionality of program routines. For system and library routines the functionality can be described in English or in a more formal notation.
- 5. Record data-flow. The recovered program structure and PDL can be analysed to identify data transformations in the software. These transformation steps show the data processing done in the program. This information is used to develop a set of hierarchical data flow diagrams that model the software.
- 6. Record control-flow. Identify the high-level control structure of the program and record it using control-flow diagrams. This refers to high-level control that affects the overall operation of the software, not to low-level processing control.
- 7. Review recovered design. Review the recovered design for consistency with available information and correctness. Identify any missing items of information and attempt to locate them. Review the design to verify that it correctly represents the program.

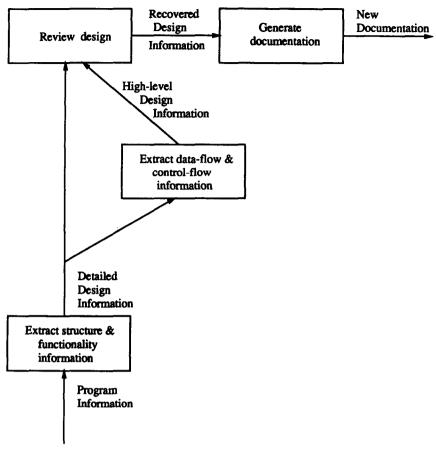


Figure 2. Reverse engineering procedure

8. *Generate documentation*. The final step is to generate design documentation. Information explaining the purpose of the program, program-overview, history, etc, will need to be recorded. This information will most probably not be contained in the source code and must be recovered from other sources.

SUPPORTING TOOLS

The only computer-based tool used during this experiment was a commercial software development environment. There are several commercial tools that claim to provide reverse engineering capabilities. In truth, such tools typically provide only program analysis capabilities that produce program-level documentation. These tools are unable to generate higher-level abstractions such as source code to PDL transformations. Most tools can generate structure charts automatically. This capability would have been useful; unfortunately, a tool with this capability was not available inhouse.

A commercially produced software development environment was used to record

the recovered design information. This environment provided an integrated collection of editors for data-flow diagrams, control-flow diagrams, structure charts, data structure diagrams, E-R diagrams, state transition diagrams and others, with an online data dictionary, and a document preparation system. This environment is designed to support software forward engineering. It does not enforce a particular development methodology, yet it is oriented towards structured design ¹⁵ with extensions for expressing the design of real-time systems. ¹⁶

A goal of this experiment was to evaluate the usefulness of this environment during reverse engineering. The needs of reverse engineering differ from those of forward engineering, but both processes need to record a design. Could an environment designed to support forward engineering be used when the work is done in reverse, i.e. starting at the most detailed level of representation with higher-level representations created later?

EXPERIENCE

During the reverse engineering effort several problems arose. Many problems concerned how to conduct the reconstruction of the design. However, the most serious problem was not detected until the design was used to create an Ada version of the program. At that point it was discovered that the design was heavily biased by the original Fortran implementation. This problem is discussed in the next section.

Collecting information

The first step was to collect information about the program and the system in general. Sources of information consisted of the program source code, one design document, a document containing an overview of the simulation system architecture, and several programmers experienced with the overall system. The reverse engineering effort was handled by the author, who had no previous exposure to this system.

Several items of information were not found. One type of missing information was data descriptions involving shared memory. The software models use signals and shared memory to communicate between themselves. Descriptions of the data passed in the shared memory areas, ranges of values and significance were not documented completely. Shared memory data items referenced by the Air Data Computer (ADC) model were briefly explained in the available design document, but other items in the shared memory were not. This meant that the recovered design, though it needed to document the structure of the shared memory area, could not document the purpose of most of the data elements. A second type of missing information was application domain information. The origin of the ADC model was not available and equations used in the software were not commented. The recovered design could list equations found in the code but could not explain their significance. Domain knowledge was needed to provide this information. Because domain knowledge was not available 'design recovery' as defined in Reference 1 was not achieved. Design information was recovered but the design could not be completely documented and explained in terms of the application domain.

Examining information sources

The second step was to become familiar with the program, its structure and how it worked. This step permitted a plan to be created for dissecting the program and entering its information into the development environment.

Unfortunately, this familiarization with the Fortran implementation also biased the reverse engineering effort. Familiarity with the details of the implementation influenced the perspective of what should be recovered and how it should be expressed.

Extracting the structure

The third step was to begin recovering the detailed design from the source code. This step involved two tasks: extracting the calling hierarchy and expressing it as a structure chart, and recording information about data exchanged between nodes in the chart. The recovered information was recorded using the structure chart editor and data structure editor in the software development environment.

To create the structure chart the main program routine became the top node in the chart. Routines called by the main routine were represented by the second layer of nodes in the structure chart. Thus, each node in the structure chart corresponded to a subroutine, function or library routine used in the program. For each node, subnodes were added if the corresponding routine contained calls to other routines. This was done until leaf nodes were reached. Leaf nodes correspond to routines that call no other routines or are library routines.

Creating structure charts from source code is easily automated. There are commercially available tools that do this. The structure charts capture the calling hierarchy of the program routines and to show their interaction. The intent was to capture this information at an implementation language independent level. Several problems were encountered while trying to achieve this language independence.

Each node in the structure chart was given the name of its corresponding routine. This allowed easy recognition between viewing the structure chart and reading the code. Unfortunately it placed a language dependency in the recovered design. Fortran restricts a routine name to a maximum of seven characters. Most names are abbreviations of more meaningful terms. Meaningful names for the structure chart nodes should have been used instead. Ideally the name selected should convey the functionality expressed by the node. This is also important when dealing with nodes that correspond to system routines. These are implementation level details. Using a more abstract and generic name in the design helps ensure that the design is free of such low-level details. For example, the name WAIT-FOR-RESPONSE is less implementation oriented than SYS\$WAIT(...).

Associating structure chart nodes with source code routines raises the issue of traceability. In reverse engineering it is desirable to record the links between the recovered design and the original source code or documentation. In this specific case it would be desirable to give a node a meaningful name and record the name of the implemented function to which it corresponds.

As the structure chart was recorded the data items passed between nodes (routines) were also recorded. Structure charts show the direction that data items are passed between nodes. Data item usage had to be analysed to determine whether data items served to carry input data, output data or both. Information about data items, such

as purpose, data type, range of values, etc, were recorded in data structure diagrams. The names recorded for data items were the names of the data items as given in the source code. The language dependency issue discussed for routine names applies here as well. More meaningful names should have been used in the recovered design.

Another problem with data items was expressing their data type in the recovered design. In an effort to have the design be language independent the Fortran data types were mapped to generic types. For example the Fortran type int2 was recorded as '2-byte-integer' in the design. A better approach would be to analyse the purpose of the variable and give it a meaningful type name. For example, a Fortran variable that contained an error number has an integer data type. When expressed in the design, a more meaningful type name would be ERROR-VALUE, where this would be an enumerated type in the design. Using meaningful design data types would have aided the clarity of the recovered design.

Common blocks posed a problem. Source routines did not use parameters to pass information. All information was exchanged via common blocks. The question was how to express this in the recovered design? A study of the common blocks and their usage revealed that there were two types of common blocks in the program. The first type of common block was used as a map to shared memory. The second type was used to pass information internally in the program.

Common blocks that mapped to shared memory had to be kept intact. The order of definitions in the common block declaration was important. This ordering was certainly an implementation detail but it placed a constraint on the recovered design since the program interacted with other programs via these shared memory areas. The recovered design had to represent the structure of these memory areas accurately. The solution was to represent these common blocks as data sources and sinks in the structure chart. The name of a common block became the name of the source/sink, and data was shown to be exchanged between the source/sink and the nodes. The data structure editor was used to record the information about these common blocks. That they were implemented as common blocks in Fortran was not recorded.

The other type of common block was used to pass data between routines in the program. It was decided that this was an implementation detail. These common blocks were broken up and their data items represented as data passed directly between routines. This would leave an implementer free to implement these items as parameters, global variables, or again as common blocks. That these data items had been collected together in common blocks was not recorded.

Recording functionality

Once the structure chart was created the next step was to record the processing associated with each node. For nodes that corresponded to system or library routines an English explanation of the routine was recorded. For source routines the processing steps were recorded using a pseudo-English PDL. This involved manually translating from Fortran to PDL.

One problem was what to do about debugging statements? The source code contained debugging statements that were guarded by conditional compilation directives. Were these part of the design? Probably not, but they were part of the operational capabilities of the software. Talks with the maintenance staff revealed

that debugging statements were never used. Therefore, these statements were not recovered

This raised the issue about how to handle conditional compilation code. Generally, such code is selected based on the operating system, machine, etc. The conditional compilation code used here referred to differences between PDP-11 systems and MicroVaxes. The maintenance staff was again consulted and it was learned that the software would not be used on PDP-11 systems, so the conditional code was not recovered. This solved the problem for this case, but leaves the general question unsolved.

A possible solution to the conditional compilation problem is to consider the purpose of the conditional compilation code. One use of such code is to handle functionality that must be implemented differently on different computer systems. Here, the functionality can be recovered and the system differences ignored as an implementation detail. A more difficult case occurs when conditional compilation code is used to hold functionality that is implemented on one computer system and not on others, i.e. system-dependent capabilities. Here, the design of the program differs depending on the intended implementation system.

Another issue that arose is where does detailed design stop and implementation begin. There is a wealth of information in the source code that would not normally be recorded in a design. The problem is realizing what should not be recovered. For example, the source code contains a system routine that requires a parameter to hold a returned status value. In the code the variable used for this parameter is of type integer. Should the system routine be included in the detailed design? Should its calling interface be recovered exactly as given in the code? Should the detailed design declare the variable as an integer or as a design level type name? In this experiment the system routines were recovered into the detailed design. Later this information was determined to be too implementation specific. A generic processing step should have been given instead.

Recovering the detailed design forced a hard look at the source code and software documentation. One error in the source code was discovered. Comments that were incorrect or outdated were also detected. The documentation was found to be badly outdated and incomplete. A total of 44 errors were located in the documentation. These errors consisted of mistakes in the documentation itself and discrepancies between the documentation and the source code. Because of these errors some information contained in the documentation could not be trusted. These errors eliminated one source of information about the software.

Recording data-flow

The fifth step was to develop a data-flow model. Structured design ¹⁵ gives guidelines for transforming data-flow diagrams into structure charts. To some extent, these guidelines can be reversed. The idea is to identify transformations on data expressed in the detailed design and use these transformations to create processes in a data-flow model. The order of transformations in the detailed design determines the order in the data-flow model.

The data-flow model consists of a hierarchy of data-flow diagrams. The top level diagram shows a single process representing the program and its input sources and output destinations. The process in the top-level diagram is then expanded in a

separate diagram. This second level diagram shows processes that together do the transformation attributed to the top-level process. Each of these processes can be expanded into lower level diagrams that show even more detailed transformations. Processes can be expanded into more detailed data-flow diagrams until atomic processes are reached.

Data-flow diagrams were constructed from the recovered detailed design using a procedure presented in Reference 17. In this procedure, data transformations in the detailed design are identified using a two-step process. In the first step, each node in the structure chart becomes a process in a data-flow diagram. The names given to processes were not the names from the structure chart nodes. Instead, descriptive process names that suggested the task of a process were used. This step creates a hierarchy of data-flow diagrams. In the second step, the PDL associated with each structure chart node is examined and data transformations are identified. Such transformations are then added to the appropriate data-flow diagram as new processes with descriptive names. In this fashion all data transformations present in the detail design are captured in the data-flow model.

Each process in a data-flow diagram had an associated process specification (pspec). A pspec gave an English description of the purpose of a process. Equations in the PDL were copied into pspecs because they showed the necessary calculation. Each data-flow diagram was also annotated. These annotations explained the significance of the data.

Recovery of the data-flow diagrams forced an analysis of the data items exchanged between nodes in the structure chart. Individual data items carry either data information or control information, or in a poorly designed system both. A data-flow model is concerned with modelling transformations on data and does not express control information. Data items used solely to carry control information were extraneous to the model and therefore were not recovered.

The higher-level abstraction of the data-flow diagrams brought out important details that were obscured in the structure charts and PDL. Two context diagrams were produced. One showed the logical environment of the ADC model and the data exchanged with other models. The second context diagram showed the actual environment of the ADC model and the data exchanged with the simulation driver program and shared memory area. The data-flow model of the ADC clearly revealed the processing structure of the program with respect to collecting data, performing calculations and communicating the results to other models. For example, a node in the structure chart that corresponded to a routine that did calculations showed calls to routines such as sqrt(), and log(). Whereas the data-flow diagram corresponding to that node clearly expressed the calculations as subprocesses with descriptive names. The data-flow diagrams emphasized what happened to the data as it flowed through the system, as opposed to the PDL where this information, while present, is buried with other details.

Creating control-flow diagrams

The sixth step was to identify the control structure of the software. The task of uncovering this structure occurred while creating the data-flow diagram. What remained was to record the control structure and explain its purpose. The control structure was recorded using control-flow diagrams, which were overlaid on the data-

flow diagrams. This step allowed both data and control-flow to be visible on the same diagram. In addition, a state control model was defined and recorded using a state diagram.

The control structure of interest for the ADC program concerned co-ordinating the actions of this program with other programs executing concurrently. This control governed access to the shared memory areas, when to read and write to them, and the exchange of signals between this program and other programs.

A problem during this step was distinguishing between low-level control structures that involved the implementation of a routine and high-level control structures that served to control the software operation. The former should be included as part of the processing described in the detailed design, the latter needs to be recorded in a control flow diagram and its control specification. The temptation is to recover too much of the control structure.

Constructing the control-flow diagram for the ADC model brought out information that was obscured in the structure charts and PDL. In addition to the control-flow diagram a state-transition diagram for the ADC model was constructed. These two diagrams clearly showed the execution control flow of the ADC program. Of course this same information was present in the structure charts and PDL, but it is not as visible in the detailed design representations.

Reviewing recovered design

At this point the recovered design was reviewed for completeness and accuracy. Reviewing for accuracy means verifying that the recovered design does describe the program, that no functionality of the program has been omitted and that no extraneous information has been recorded. The point to keep in mind is that this effort was to document the recovered design, not the program.

Reviewing for completeness involves determining what information is missing and whether it can be recovered. It was not possible to recover the design completely. The most difficult information to recover was the reason behind a processing step. Other information that could not be recovered was:

- 1. Equations used in the code were not documented. The significance of these equations could not be determined without knowledge of the application domain.
- 2. The meaning of the information held by some data items was not documented. Possible significance could only be guessed at.
- 3. The range of possible values for some data items was not known and could not be determined.

Document generation

The final step was to prepare the recovered information and format it into a design document. Information not recovered in earlier steps was collected at this point. This included an overview of the system, an overview of the program, its intended purpose, history, related documents, etc. The software development environment used provided a document generation system. This system was used to generate a design document for the recovered design.

GENERATING AN ADA IMPLEMENTATION

The recovered design was used to develop an Ada implementation of the ADC program. Because the design had been recovered from a Fortran program, using a different language to reimplement the program served to test the adequacy of the recovered design. This step revealed many of the shortcomings of the recovered design that were discussed in the previous section.

The software development environment used to record the design was used to generate Ada data declarations from the data structure diagrams. The shared memory area data structure chart was converted into Ada data declarations and then placed in a package declaration. Structure chart nodes were converted into Ada procedure declarations using an Ada procedure declaration template. The associated PDL was inserted into the procedure body. The PDL was then manually translated into Ada.

It was during the translation of the PDL into Ada that the Fortran implementation dependencies were noticed. A fundamental property of software designs, as they are normally created in forward engineering, is that a design should be free of constraints imposed by an implementation language or the implementation phase. An important problem in reverse engineering is recognizing effects of implementation phase decisions on the program. Such decisions are imposed on an implementor by the implementation language, the operating system, the machine architecture, etc. However, the effects of these decisions should not be recovered in the design. Identifying or unraveling such effects proved to be difficult.

For example, the Fortran code contained a call to a system function that returned a status value when completed. The calling interface of this system function was recovered. The data type of the status variable was integer; this plus the purpose of the variable was recovered. In Ada, the same system function had a different name and interface. It was no longer a function but a procedure, and the status variable was passed as a parameter. The status variable no longer had a type of integer. Its Ada type was COND_VALUE_TYPE. In addition, the value of the status variable was checked in several IF statements. In Fortran the condition tests had the form if (variable) then Because the Ada version of the variable was not of type boolean the condition tests in Ada had to be rewritten to check against a specific value.

This example points out that data type information should be represented in an abstract way that emphasizes the purpose of the data as much as its range of values. During the implementation phase the decision can be made about which language-supported data type will be most suitable to express a design-specified data type.

The strong Fortran bias in the recovered PDL could have been reduced by creating a new structure chart and PDL from the recovered data-flow and control-flow models. These models were further removed from the original source code and were less biased by the implementation than the recovered detailed design. By recreating the detailed design from the recovered high-level design, a new language-independent detailed design could have been produced. However, this is equivalent to redesigning the program, which may not be desirable. It is more desirable to do a better job of recognizing implementation details and not recovering them.

Another issue raised by the recovered design was the effect of efficiency concerns during the implementation phase. Programmers often write complicated code sections to achieve better run-time performance. During design recovery these complicated code sections affect the recovered design. The problem is to identify such code

sections. One solution is to restructure these code sections to be cleaner and easier to understand. The question is when should this be done? Should it be before recovery, during recovery, i.e. not changing the current implementation, but recovering a cleaner design, or after the recovery process has been completed? The issue was not resolved.

CONCLUSIONS

This paper has presented a procedure for reconstructing the design of an existing software system. This procedure was used to reconstruct and document the design of a program. The recovered design was recorded in a software development environment and used to develop a new implementation of the program. The experiences and observations gained during this work have been described in the preceding sections. Several issues have been identified and these are summarized in the following paragraphs.

- 1. Implementation bias. Perhaps the most important issue in recovering a design is to separate design information from implementation information. At the implementation level it can be difficult to recognize what information should not be recorded in the recovered design. It can be difficult to 'throw away' information during the abstraction process, particularly, when the goal is to reimplement the system. It is necessary to realize that the design does not serve as program documentation. Program documentation reflects the implementation details of the source code. Design documentation reflects a higher-level view of 'how' a program functions. Implementation details that need to be watched for are data item names, basic data types, data structuring, system interfaces, code sections written to be efficient, code that is biased by the underlying computer architecture and code whose expression was biased by the original implementation language.
- 2. *Traceability*. A recovered design should record links between recovered information and the original sources. This permits traceability between the recovered design and the original implementation. This experiment has shown that an existing software development environment can be used to record a recovered design. However, such a tool must provide the flexibility to record information not envisioned when the tool was created.
- 3. Domain information. The information recorded in a program is sufficient to explain 'what' is being done, but not 'why'. To understand the significance of a processing step, deeper information is often required. This information is often not recorded in the source code or existing documentation. An understanding of the application domain can aid the recovery of information about the purpose of a function and its significance.
- 4. Re-engineering. Reverse engineering will most often be done on older software systems whose documentation is out-of-date or non-existent. Such systems will often be candidates for re-engineering: rewriting the system to improve its understandability, ease maintenance, remove dead code, etc. It is easier to change a design than source code. When should design changes occur: as the design is recovered or afterwards? It seems reasonable to change the recovered design. This way the recovered design describes the current system. The recovered high-level design will be updated and a new detailed design generated.

A new version of the system generated from this design can be developed using modern techniques and programming languages. This will result in a system that is better structured, documented, and more easily maintained than the previous version. Thus, reverse engineering as a part of re-engineering will prove to be useful in extending the life of older programs.

5. Existing documentation. One last issue is the use of comments and existing documentation. These sources of information may be out-dated or incorrect. They must be used carefully. They can provide useful information about a program, but they can also be misleading or wrong. The source code is the final statement about what the program actually does.

In summary, this experiment was successful in identifying several issues in software reverse engineering. It is hoped that further investigation into these issues will result in a methodology for software reverse engineering and aid in the development of took to support this task.

ACKNOWLEDGEMENTS

Research sponsored by the Air Force Office of Scientific Research/AFSC, United States Air Force, under Contract F49620-88-C-0053. The United States Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation here on.

REFERENCES

- 1. E. J. Chikofsky and J. H. Cross II, 'Reverse engineering and design recovery: a taxonomy', IEEE Software, 7 (1), 13–17 (1990).
- 2. S. C. Choi and W. Scacchi, 'Extracting and restructuring the design of large systems', IEEE Software, 7 (1), 66-71 (1990).
- 3. Y. F. Chen, M. Y. Nishimoto and C. V. Ramamoorthy, 'The C information abstraction system',
- *IEEE Trans. Software Engineering,* **SE-16,** (3), 325–334 (1990).

 4. J. R. Cordy, N. L. Eliot and M. G. Robertson, 'Turing Tool: a user interface to aid in the software maintenance task', *IEEE Trans. Software Engineering*, **SE-16**, (3), 294–301 (1990).

 5. J. A. Ricketts, J. C. DelMonaco and M. W. Weeks, 'Data reengineering for application systems',
- Conference on Software Maintenance, Miami Florida, 16-19 October 1989, pp. 174-179.
- 6. T. J. Biggerstaff, 'Design recovery for maintenance and reuse', Computer, 22 (7), 36–49 (1989).
- 7. I. Somerville, Software Engineering, 3rd edn, Addison Wesley, 1989.
- 8. R. N. Charette, Software Engineering Environments: Concepts and Technology, Intertext Publications, Int, New York, 1986.
- 9. R. A. Converse and M. J. Bassman, 'Conversion to Ada: does it really make sense?', Avionics Panel Symposium: Software Engineering and Its Application to Avionics, AGARD-NATO, Cesme Turkey, 25–29 April 1988, paper 8.
- 10. R. C. Waters, 'Program translation via abstraction and reimplementation', IEEE Trans. Software Engineering, 14 (8), 1207–1228, (1988).
- 11. P. J. L. Wallis, 'Automatic language conversion and its place in the transition to Ada', ACM Ada Letters (Proceedings of the Ada International Conference, Paris France, 1985), V (2), 275–284,
- 12. H. M. Sneed, 'Software renewal: a case study', IEEE Software, 1(3), 56-63 (1984).
- 13. R. N. Britcher and J. J. Craig, 'Using modern design practices to upgrade aging software systems', *IEEE Software*, **3** (3), 16–24 (1986).
- 14. P. Antonini, P. Benedusi, G. Cantone and A. Cimitile, 'Maintenance and reverse engineering: low-level design documents production and improvement', Conference on Software Maintenance, Austin Texas, 21–24 September 1987, pp. 91–100.
- 15. E. Yourdon and L. L. Constantine, Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design, Prentice-Hall, 1979.

- 16. D. J. Hatley and I. A. Pirbhai, *Strategies for Real-Time System Specification*, Dorset House Publishing, 1987.
 17. P. Benedusi, A. Cimitile and U. De Carlini, 'A reverse engineering methodology to reconstruct hierarchical data flow diagrams for software maintenance', *Conference on Software Maintenance*, 16–19 October 1989, pp. 180–189.