REENGINEERING USING A DATA ABSTRACTION BASED SPECIFICATION LANGUAGE

by

Randall E. Duran

August 1991

Submitted to the Department of
Electrical Engineering and Computer Science
in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1991

Signature of Author [illegible]
Randall E. Duran
Department of Electrical Engineering and Computer Science, Sept. 1991

Certified by [illegible]
Professor John V. Guttag
Thesis Supervisor

Certified by [illegible]
Phil Hwang
Company Supervisor

Accepted by__________
Campbell L. Searle
Chair, Department Committee on Graduate Students

Table of Contents

Table of Figures

Abstract

This paper describes how a reengineering methodology was developed and applied, to convert part of a system implemented in CMS-2 to a new C implementation. In particular, this methodology used Larch specifications as an intermediate design representation that was obtained through reverse engineering. The benefits of using this methodology and ways of improving it, primarily through automation, are suggested.

CHAPTER ONE

INTRODUCTION

Systems reengineering is the practice of taking an existing system and reimplementing it in a new and more advantageous form. This technique avoids the expense of completely redeveloping system software by allowing it to be transitioned to a more modern and maintainable language using its existing design. This process may also offer the benefits of extending the life of the system, reducing hardware dependency, improving maintainability, and generating new and more accurate documentation.

Little formal research has been performed on the reengineering of software from an existing system into a different programming language. The research to date has focused on redocumentation [3,29] and restructuring [5,27]. While these efforts have been useful, they have not fully addressed the problem of language to language software reengineering.

It was the intent of this research to experiment with a particular reengineering process and methodology, and determine its effectiveness when used on source code from an existing system. Although the scope of this project only allowed a fragment of a software system to be reengineered, this research did provide a foundation for understanding the process, results, and challenges of reengineering. The intent was that future research would extend from this research, so as to create a more complete characterization of the reengineering process.

The research focused on the reengineering of a 299 line CMS-2 procedure and the data objects that it used to a C implementation, using Larch specifications as an intermediate representation. The unclassified CMS-2 source code was taken from an existing U.S. Navy weapons system, the MK116 Mod7 torpedo system. From this work, a reengineering methodology was developed, results were obtained from an experimental application of this methodology, and information regarding possible automation of the reengineering process was gained.

The rest of this thesis is organized as follows: Chapter Two defines relevant terms and provides background information about the reengineering project; Chapter Three provides specific information about the languages and systems used in the project; Chapter Four describes the part of the reengineering methodology that addresses reverse engineering; Chapter Five describes the forward engineering process; Chapter Six presents interesting examples encountered during reengineering and analyzes the project's results; Chapter Seven provides the research's conclusions and suggests areas for future study.

CHAPTER TWO

THE BACKGROUND OF SYSTEMS REENGINEERING

Systems reengineering is a research area that integrates maintenance and new development. The large number of systems in the maintenance phase of their lifecycle along with the advances in development techniques has made it profitable to focus on improving current systems using new development methods. The upgrading, or traditional maintenance, of existing systems often costly because of problems with its implementation and prior maintenance practices. Often the size and complexity of existing systems makes it difficult to redevelop software to help reduce maintenance costs. System reengineering, using an intermediate representation, may offer a cost effective alternative to redevelopment to attain improved system maintenance. In particular, the use of formal methods as an intermediate representation may be helpful by providing accuracy and precision.

2.1 Definitions

This section defines the terminology used throughout the rest of this paper. Most of the definitions are commonly accepted, but some definitions have been extended so as to convey a meaning more relevant to this research. They move from general reengineering definitions to definitions more specific to this project.

Figure 1 provides the generalized definitions for several reengineering

Design Representation - The implementation-independent, abstract representation of the design of a system.

Forward Engineering - The process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system.

Redocumentation - The creation or revision of a semantically equivalent representation of a system intended for a human audience.

Reverse Engineering - The process of analyzing a subject system to identify the system components and their interrelationships, and to create representations of the system in another form or at a higher level of abstraction.

Reengineering - The examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form. This includes some form of reverse engineering (to achieve a more abstract description) followed by some form of forward engineering or restructuring.

Reimplemenation - The process of forward engineering a design representation which was obtained through reverse engineering.

Restructuring - The transformation from one form of system representation to another at the same relative abstraction level while preserving the subject system's functional requirements.

Figure 1. Generalized reengineering definitions.

related terms, which are based on those of Chikofsky and Cross [6]. These definitions could apply to any number of types of systems including hardware, software, organizational, etc. The definitions shown in Figure 2 apply to reengineering at the system level [21]. They address the integrated components of systems and do not view system as a single unified entity. Figure 3 provides definitions that are specific to the reengineering of software systems. These definitions focus on the components of a software system and the specifics of the software reengineering process.

Environmental Characteristics - The external properties which affect the design of an embedded system including human-machine interfaces and the organizational structure which the embedded system is a part of.

Nonfunctional Requirements - The real-time, time critical, reliability, fault tolerance, maintainability, and security issues which characterize systems.

Systems Reengineering - Transformation of any of the components in a system including the hardware, software, and environmental characteristics which produces a new implementation of the system.

Figure 2. System reengineering definitions.

Software reengineering, as defined, will be simply referred to as reengineering henceforth. Thus, the term reengineering will refer to 1) the transformation from one programming language to another 2) the use of an intermediate design representation, 3) the reverse engineering process, and 4) the forward engineering process. The latter two points are inherited from the generalized definition of reengineering. These four points are important for characterizing the type of reengineering addressed in this paper.

2.2 Reasons for and Benefits of Reengineering

Software reengineering is an important issue because of the large number of existing systems that are implemented in older programming languages. These systems have severe maintenance problems because of a

Software Reengineering - The reengineering of an existing software system to new target language using an intermediate representation to capture the software design information.

Source Language - The programming language that an existing software system is implemented in.

Target Language - The programming language that is used to represent a forward engineered software implementation.

Translation - The automated conversion of software written in one programming language to another programming language using direct syntactic substitution.

Figure 3. Software reengineering definitions.

lack of portability, poor documentation, and poor prior maintenance practices. It is also difficult to hire new programmers and maintainers that are familiar with some older programming languages.

Through the exclusion of implementation specific information in the design abstraction process, hardware dependencies in reengineered software may be reduced. The removal of these aspects improves portability between hardware platforms, allowing more flexible use of new hardware technology. Removal of hardware dependencies also allows the maintainer who is unfamiliar with a system's hardware to understand the system's functionality more easily.

The reengineering process extracts software's design into a representation that can be used as documentation for the old system as well as the new. Since the documentation is derived directly from the software, it is likely to be both consistent with the software and up to date. This method of generating of new documentation can improve understanding of the system by providing information that was previously undocumented (i.e., changes that have been made, the relationship between different modules, and system complexity statistics). This information can be useful for determining how future maintenance should be approached.

The ability to use new maintenance environments can also be gained through reengineering. By porting software to new languages and systems, new maintenance tools, which could not be used with the previous language or system, can often be used. Such tools can include new CASE, automated documentation, and software analysis tools.

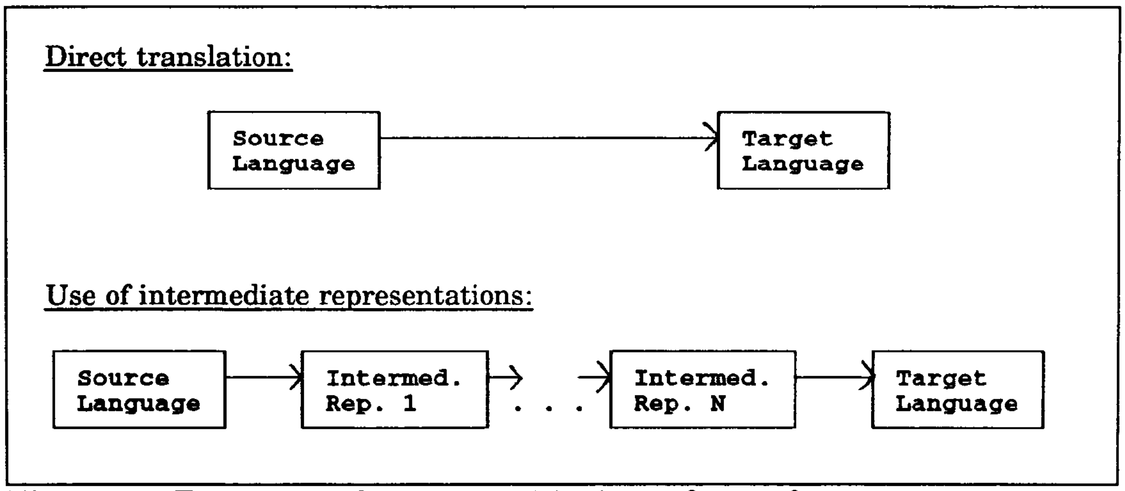

Figure 4. Two approaches to transitioning software from one programming language to another.

2.3 Reengineering Strategies

Two approaches can be used to transition software from one programming language to another: 1) the use of direct translation from one language to another, and 2) the use of intermediate forms to represent the software in the transition between the two languages (see Figure 4). These two approaches will be referred to as the *translation and restructuring approach* and the *intermediate representation approach*, respectively. The direct translation approach is often enhanced by automated restructuring of the new source code, so that it can take advantage of the target language's characteristics.

While restructuring [5,27] is a key point in the first approach, it is important to note that restructuring is a process that can be applied whether or not an intermediate representation is used. Also, not only can the reimplemented source code be restructured, the original source can be restructured prior to any transitioning.

2.3.1 The Translation and Restructuring Approach

The focus of the *translation and restructuring approach* [4] is on providing effective translation and useful restructuring. One advantage that this approach offers is that the translation process can be simple if similar languages are used. Another advantage of this approach is that much of the focus is on using and developing restructuring and repackaging tools, which can also be applied to traditional software development practices.

Two disadvantages of this approach are that it does not necessarily improve understanding of the system, and inefficient implementations in the original code may persist in the new implementation. These disadvantages occur because reverse engineering is not employed in the transitioning process. Because the restructuring is automatic, and is based on the new source code, no reverse engineering is performed and thus a design abstraction is not obtained. This, in turn, leads to no improvement in system understanding. Similarly, because there is no functional abstraction, many of the previous implementation decisions will migrate to the new implementation.

2.3.2 The Intermediate Representation Approach

The *intermediate representation approach* is what this thesis refers to more specifically as software reengineering [9,23]. This approach focuses on using a series of transformations, which result in one or more intermediate forms. The emphasis is on applying transformations that will improve maintainability and result in improved implementations.

The advantages of the *intermediate representation approach* are that it provides design level documentation for the system, allows restructuring at the design level, and abstracts away from implementation dependent issues. The use of intermediate representations provides an automatic form of documentation for both the original system and the new implementation. Likewise, these intermediate forms provide a framework for restructuring at a more abstract level than that of either the original or the new implementation.

The disadvantages of this approach are that the process can be more complex and may not be as easily automated as the *translation and* *restructuring approach*. The main difficulty is that abstraction away from the source code into intermediate representations requires complex analysis, which can not be fully automated.

2.4 Research Approach

The approach taken to this project was to define a high level reengineering process, use it to experimentally reengineer part of an existing system, and define a reengineering methodology based on this experience. Defining a high level reengineering process involved choosing a primary intermediate representation and determining what major steps were involved.

Although the approach was accomplished primarily using manual methods, the reengineering process must be partially automated to be effective. In the course of experimenting with this process and determining a methodology, it was useful to consider what parts might be automated. This involved observing what parts were mechanical, what parts required analysis of large amounts of data, and what parts could be improved by providing a better reengineering environment. While it was not feasible to develop the necessary automation as part of this research, it was possible to capture the reengineering process and the methodology's fundamental needs for automation.

CHAPTER THREE

COMPONENTS OF THE REENGINEERING PROJECT

3.1 The Source Language: CMS-2

For the reengineering example, it was necessary to choose a source language for which could benefit from reengineering, and which contained interesting systems characteristics. The CMS-2 programming language [2] was chosen as a source language because it fit these requirements. There was a large amount of CMS-2 software that was in the maintenance stage and that could be reimplemented in another programming language to improve portability and maintainability. CMS-2 also contained many systems related characteristics such as hardware dependencies, real-time performance requirements, reliability requirements, and interesting intermodular characteristics.

3.1.1 Background

CMS-2 was developed in the late 1960s for the Navy, for use in the development of tactical systems. Software written in CMS-2 is intended to be run on various AN/UYK and AN/AYK series computers. There are several dialects of CMS-2, including CMS-2M and CMS-2Y, which correspond to the different hardware platforms on which they are designed to be run. Nearly all Navy tactical systems have been implemented using CMS-2 or a dialect thereof.

3.1.2 Structure

CMS-2 programs are divided into data areas, executable statements, and control information areas. Data can be defined either locally or globally (at the system level). Executable statements are contained in functions and procedures, which can be reentrant. Control information, consisting of compiler switches, macro substitutions, and hardware dependent information, is included in header blocks.

3.1.3 Features

One of the more interesting aspects of the CMS-2 language is how data is organized and manipulated. CMS-2 provides specific control over how data is allocated and stored, by allowing the user to specify how tables are arranged and packed. The overlay feature in CMS-2 allows the same data area to be referenced by different variable names and various types. The addresses of data objects can also be determined during run time and used in the executable code.

The ability to use embedded assembly language within CMS-2 source code, referred to as direct code, also gives CMS-2 programs interesting properties. The DIRECT keyword is used to define a code block that contains macro assembler code that interacts with the CMS-2. Such code blocks are primarily intended for I/O operations. Direct code blocks can be used for data declaration as well as execution of operations. Using direct code usually requires an understanding of how registers are being used by the CMS-2 code that surrounds the direct code block. In addition, understanding how registers are used in CMS-2 procedures often requires the compilation of the surrounding CMS-2 code and the examination of the object code generated.

CMS-2 does not explicitly support data abstraction and provides little isolation from the underlying hardware. Many of CMS-2's attributes result from the era in which the language was designed. In the late 1960s both memory resources and computational speed were severely limited; it was important to be able to closely interact with the hardware so as to take full advantage of the resources. These considerations are reflected in the data allocation and manipulation capabilities, and the direct code feature.

3.2 The Intermediate Representation: Larch Specifications

Software reengineering can use many different types of intermediate representations. Possible forms include data flow diagrams, abstract grammar trees, formal specification languages, program design languages, and informal descriptions. Each of these representations have their benefits and drawbacks. In choosing one or more intermediate representations for reengineering, it is important to consider what purpose the representation must serve.

Given the objectives of the research, it was necessary to find an intermediate representation that was accurate, precise and flexible enough to robustly handle systems issues. The Larch family of specification languages was chosen as an intermediate representation because of its features and the available support. Larch specifications were flexible and could be tailored to describe software written in various programming languages. Larch also supported both intermodular and intramodular specification well, which was necessary for a design representation to be effective. The availability of semantic and syntactic checking tools for Larch also made it attractive.

3.2.1 Structure

Larch is a formal, two-tiered set of specification languages. The first tier, the Larch Shared Language (LSL), provides a programming language independent form for defining terms used in interface specifications and generating theories about those terms. The second tier, the Larch Interface Languages (LILs), specify what is needed to write and use program modules. The LILs contain language specific information about data representations, module interfaces, and exception handling.

This two tier division is advantageous for reengineering because it allows a common form of representation, the LSL description, as well as a language specific form, the LIL interface. There are multiple LILs currently available including Larch/C, Larch/Ada, and Larch/Modula-3 [10,14]. This division allows target languages to be changed by only respecifying the LIL part of the specification.

3.2.2 Features

Larch specifications are based on the use of data abstractions. This specification style provides an understanding of program functionality based on the data constructs as well as the control and data flow. By reverse engineering software into data abstraction-based specifications, new implementations can take advantage of the encapsulation and data hiding that result from the specification process.

Incremental construction is also a key feature of Larch specifications that make them both more understandable and easier to compose. Specifications are composed by extending and constraining preexisting specifications to form new ones. This technique results in the building of a hierarchy of specifications. Such a hierarchy abstracts information out of the individual specifications making them more easily understood. Likewise, the division of specifications into multiple pieces encourages reuse because existing specifications can, and should be used as a basis for new specifications.

For this project, the Larch/C interface language (LCL) was used in conjunction with the LSL to represent the software's functionality. Support provided for Larch included LSL and LCL checkers. The LSL and LCL checkers provided automated semantic and syntactic checking. The Larch Prover [11,12], which performs logical checks on specifications, was not used because of the time and resource limitations. Ideally, LSL and a Larch/CMS-2 interface language would have been used to express the intermediate representation attained through reverse engineering. LCL was used, though, because Larch/CMS-2 did not exist and could not be developed within the scope of this project.

3.3 The Target Language: The C Programming Language

Reengineering software required that the target language be flexible, easily maintainable, widely supported, and easily portable. The C programming language was chosen as a target language because it met all of these requirements: it was flexible in that it supported both high and low level programming needs; it was simple, it was designed with systems programming as a primary consideration; and it was popular. The available support for the Larch/C interface language also made it a good choice.

3.3.1 Background and Structure

The C programming language was developed in the early 1970s by Dennis M. Ritchie. It was designed as a general purpose programming language that is highly portable. C is a fairly low level language and relies heavily on the use of library functions. C uses libraries to avoid hardware dependencies that occur when hardware dependent functions are included in a programming language. The C programming language has been standardized by the American National Standards Institute (ANSI) [1,17].

3.3.2 Features

C provides pointers and the ability to do address arithmetic, both of which are used extensively in C programs. A preprocessor allows for macro substitutions and compiler switches. Standard C libraries provide many of the I/O, string handling, and math functions that are built into other languages. C also allows explicit control over run-time memory allocation and deallocation.

C provides great flexibility because of its low level nature, and is easily understood because of it's simplified syntax. It does not include explicit mechanisms for supporting data abstractions. Reimplementing CMS-2 software in C can allow simpler expression and improved portability over the original implementation.

3.4 The Test Case: The ASWCS 116/7

The reengineering test case had to be part of an existing system in the maintenance phase of its life cycle. It was also important that the system could benefit from reengineering and had interesting systems characteristics. Part of the ASWCS 116/7 software was chosen as a reengineering test case because it fit these requirements well. The ASWCS 116 system had been in the maintenance phase for several years and had just undergone a revision. It was bound to specific hardware because of its CMS-2 implementation. It was also a mission critical computer system, which meant that it had real-time performance requirements and fault tolerance needs.

3.4.1 Background

ASWCS 116/7 software is a decision support system for a torpedo weapons system. It integrates the inputs from external sensors and user controls, analyzes the information, and displays relevant information to the operator. It consists of approximately 200,000 lines of CMS-2 code and is divided into 5 major subsections. The ASWCS 116/7 system was based on the previous ASWCS 116/5 system, which was also written in CMS-2, and has been under development for approximately three years. The ASWCS 116/7 system is just beginning its field use.

Many of CMS-2's characteristics make the ASWCS 116/7 software difficult to maintain. Lack of modularity and clearly defined interfaces are two predominant maintenance obstacles. These problems require the understanding of a large portion of the system to maintain only a small section of it. The limitation of eight character symbol names, which CMS-2 imposes, also makes it difficult to create meaningful names in a large system such as the ASWCS 116/7. The use of in-line assembly language programming and overlayed memory are additional factors that make understanding the software difficult.

While the Navy uses the software and is responsible for maintenance, the Navy did not develop the ASWCS 116/7 software. Rather, the Navy provided the functional requirements for the system and it was developed by General Electric. There are several forms of documentation available for the ASWCS 116/7 software system. Three of the primary forms are: the Program Performance Specification (PPS), the Program Design Specification (PDS), and the Program Description Document (PDD). The PPS was the functional requirement specification on which the program development was based. The PDS was the design level specification that provided a mapping between the PPS and the program modules. The PDD consisted of a low level description of the function of each module and was expressed in a program design language (PDL).

The documentation is difficult to understand because of its large quantity (several shelves worth), the extensive interrelation between the program sections, and the relationships that must be traced between the various documentation levels. Also, much of the documentation is inconsistent with the actual software because the implementation of many sections was deferred. Using this documentation and comments provided in the software, Navy personnel must correct programming errors, implement new features, and otherwise maintain the system.

3.4.2 Structure

The ASWCS 116/7 software consists of four major subprograms: the Auto Processing Computer Program (APCP), Display and Control Computer Program (DCCP), Mission Support Computer Program (MSCP), and Executive Support Computer Program (ESCP). The functions of the Mission Support Computer Program are to support display and control, provide system readiness assessment, provide readiness control, and handle data extraction and recording. Of these functions, the readiness control (RC) function provides the ability to initiate, maintain, and terminate the overall operation of the system. The RC function provides initialization control, termination control, system recovery and mode control.

The RCININIP subtask, part of the RC function, performs initialization control and system recovery for the MSCP. This subtask is executed by four subtask units: RCININIT, RCINPERD, RCINSUCC, and RCINMGT. RCININIT serves as the initialization entrance control; RCINPERD serves as the periodic entrance control; RCINSUCC provides the successor entrance control; and RCINGMT processes the entry of the time and date.

The RCINSUCC subtask unit is a task that consists of two procedures: RCINSUCC and RCINFAIL. The RCINSUCC procedure processes the successor entrance requirements for initialization processing. RCINFAIL handles the processing for initialization response failures. Of the subtask units and procedures available, the RCINSUCC procedure of the RCINSUCC subtask was chosen to be a reengineering example. The choice of the RCINSUCC procedure was somewhat arbitrary, but was based primarily on the size and complexity of the module. The listing for RCINSUCC can be found in Appendix A. The source code itself is fairly difficult to understand because of the enigmatic variable names and lack of modularity.

3.4.3 Details

The RCINSUCC procedure is an average sized procedure in the ASWCS 116/7 system that consists of 521 text lines of code, of which the first 222 lines are PDL comments and following 299 lines are actual source code. RCINSUCC has no formal parameters passed to it, and there are no local variables for the RCINSUCC procedure other than two index variables that are used as loop counters. The data used by the procedure, 12 tables and 25 variables, is contained in separate database files containing the data declarations. The database files correspond to subtask, task and global level data. The ASWCS 116/7 software also uses the features of the SDX/SDEX operating system extensively, including task scheduling.

CHAPTER FOUR

REVERSE ENGINEERING

4.1 Introduction

Reverse engineering was by far the most extensive and challenging part of the project. Most of the effort included understanding the CMS-2 code and finding ways of describing its functionality using data abstractions. Although the reverse engineering process was accomplished manually, automated analysis and support techniques would benefit this process.

4.1.1 Goals

The goal of the reverse engineering process was to provide a design representation that would improve maintainability. Based on this goal, the process was oriented towards providing conceptual information about the design that would improve understanding and thus improve maintainability of the new implementation. This goal also made the reverse engineering process more complex and interesting. The process could have been simplified, but it would not have provided as effective a study or as useful results.

In particular it was the objective of reverse engineering to create an intermediate design representation that abstracted design information about the software into a more usable form. This form would be more conceptually understandable than the implementation, show relationships that were not apparent in the implementation, and hide implementation issues. It was also important to have available a mapping between the intermediate design representation and the original implementation, so that both forms could easily be compared.

4.1.2 Approach

Improvement in maintainability was achieved through an extensive reverse engineering process. This process involved symbol renaming, data reorganization, module decomposition, and specification of sequential dependencies. Through these phases a procedural representation of the original CMS-2 source code was transformed into LCL procedural specifications and the data abstraction, which were identified, were described by LSL data specifications. The final result was a set of Larch specifications that represented the functionality of the original software. It is important to note that the lack of a Larch/CMS-2 interface language was a primary factor that led to the use of this approach.

The reverse engineering process was performed in six phases: renaming, module decomposition, data analysis, data recomposition, dependency determination and specification, and conversion into LCL specifications. Renaming was used as an initial measure to make the reengineering process and resulting specifications more comprehensible. Module decomposition was based on complexity and module structure, and was performed to make the procedural representation more manageable. Data analysis examined the functionality and interrelation of data and determined how new data abstractions should be formed. Data recomposition formally defined the data abstractions in the form of LSL specifications. Dependency determination and specification analyzed the source code to identify sequential relationships, and then specified them in the procedural representation. The final stage of the reverse engineering process was the conversion of the procedural representation into LCL specifications.

A representation of the original CMS-2 procedure was transformed during each stage of the reengineering process. This representation was referred to as the procedural representation of the original software module. The procedural representation began as the original CMS-2 source code. Over each phase of the reverse engineering process, the procedural

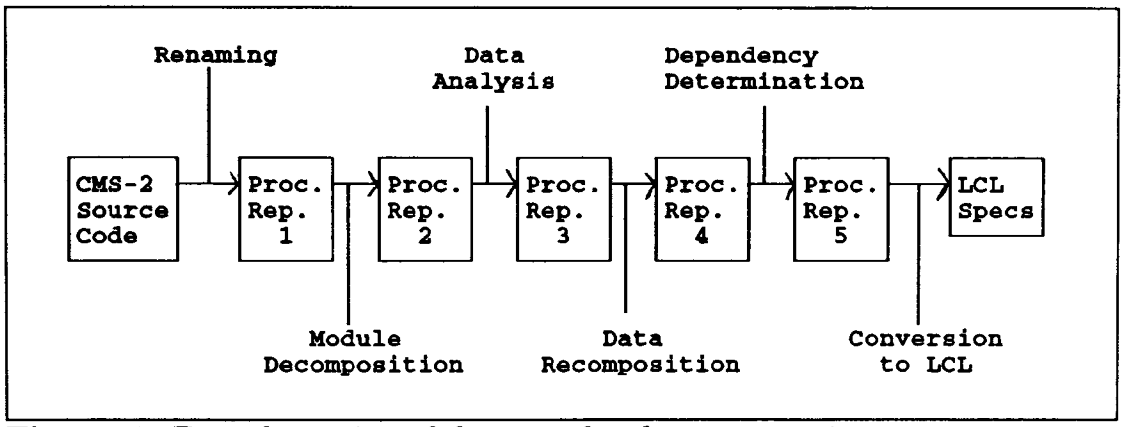

Figure 5. Transformation of the procedural representation.

representation came closer to resembling the final LCL specifications that were generated, as shown in Figure 5.

The lack of a Larch/CMS-2 interface made the reverse engineering process more difficult, because it was necessary to partially describe the CMS-2 code using LCL specifications. Ideally, the reengineering process would have reverse engineered the CMS-2 software to a LSL and Larch/CMS-2 specification set. This interface language would provide a more natural means of specifying an interface from the LSL specifications to the CMS-2 code. Then, as part of the forward engineering process, the CMS-2/Larch specifications would be transformed to LCL specifications so that C source code could be written using the LCL specifications. Since there was no Larch/CMS-2 language available and would be difficult to develop as part of this project, it was necessary to respecify the CMS-2 directly into LSL/LCL.

4.2 Renaming

The renaming phase of reverse engineering was one of the most straightforward and useful parts of the reverse engineering process. This phase involved choosing more descriptive names for the CMS-2 variables, reexpressing variables as new types that better characterize the data's function, and reexpressing the procedural representation of the source code in a form that was more easily understood by the researcher. These changes would eventually propagate to the Larch specifications and sometimes to the new implementation. When renaming, it was necessary to keep track of how the original source code names and structures mapped to the new procedural representation and vice versa.

The only symbols that were not renamed were other functions, which had not been reverse engineered, and system functions. It was intended that these symbol names would be changed at a later point in the reengineering of the system. Only parts of the system that had been reengineered would have their symbol names changed. This would make it evident which parts of the system had not been reengineered yet.

4.2.1 How Renaming Was Performed

The CMS-2 data variables were renamed and CMS-2 keywords and data variables were reexpressed. Some simple analysis was required to reexpress data types, so as to achieve more effective renaming. Reexpressing the CMS-2 keywords was helpful for making the procedural representation more generic and thus more easily understood.

4.2.1.1 Data Renaming

Data objects were renamed based on their function. Variable of simple types were renamed as single variables with new names. When complex data structures were renamed, their subcomponents were renamed also. The names chosen were primarily determined by comments around the data declaration and comments near references to the data. The names were also partially determined by simple analysis of how the data was used.

An example of how a simple variable was renamed is shown in Figure 6. The variable QTMFG is declared as an integer, but the comments accompanying the declaration suggest that the variable is actually used as a based on the declaration:

```
''CRITICAL TASK FAIL FLAG, 0/1''
VRBL QTMFG     I 32 S $

QTMFG                    is renamed as ->          crit_task_fail_flag
```

Figure 6. An example of data renaming.

flag that denotes a critical task failure, and that it will only hold the values 0 and 1. Upon tracing of all references to that variable in the program, it was determined that the variable was indeed only assigned values of 0 and 1. Based on this knowledge the variable was renamed as a flag.

4.2.1.2 Data Reexpression

Simple data analysis was also performed at this stage of the reengineering process so that the data could be reexpressed as simpler types. Types were simplified where possible, so as to make their functionality more apparent. The example above showed how an integer type is determined to be equivalent to a boolean in its use, and its renaming based on this information. Similarly, some tables could be reduced to tuples, or structure types, and some variables could be reduced to enumerated types. This analysis was useful as a basis for more complex data analysis at later stages of the reverse engineering process.

An example of such an equivalence is shown in Figure 7. The table QMTERMRQ has only one dimension and therefore is equivalent to a tuple or structure type, which is how it is reexpressed. The introduction and use of partially based on the declaration:

```
TABLE QMTERMRQ  V 1 1          ''TERMINATION (TERM) MESSAGE''$
      FIELD REQ I 8 U  0  7        ''TYPE OF TERMINATION -
                                    OPERATOR REQUEST OR
                                    RCIN, 1 - 2'' $
      FIELD FAILTYPE I 8 U 0 15    ''FAILURE TYPE,1 - 2''$
END-TABLE QMTERMRQ $
```

```
QMTERMRQ(0,REQ)          is reexpressed as     term_msg.term_type
QMTERMRQ(0,FAILTYPE)            ->             term_msg.fail_type
```

Figure 7. An example of data reexpression.

a simpler type is useful in this situation because it enables the data's type to better reflect the data's function. After renaming and reexpression, it was obvious by viewing the procedural representation that QMTERMRQ was a message structure. This was not apparent from the reviewing CMS-2 procedure.

4.2.1.3 Keyword Reexpression

CMS-2 keywords were also reexpressed and the structure of the procedure representation was modified to accommodate the new expression. Reexpression was used to move away from CMS-2 specific control statements and structures such as a *FOR* statement, which is the equivalent of a *switch* statement in C, to more generic and LCL-like forms such as an *if*, *else*, and *else if* representation. The indentation structure was also changed at this stage to assure that the indentation was consistent throughout the procedural representation and to make the control blocks more clear. It is important to note that if a Larch/CMS-2 interface language was being used instead of the LCL or if researcher was more familiar with CMS-2 this process may not have been necessary or as useful.

4.2.2 Substitution

After new names were chosen and the means of reexpression was determined, the new forms were substituted into its procedural representation for the original symbols. Also, comments, blank lines, and *BEGIN* and *END* statements were removed from the procedural representation. Removing these constructs resulted in an 40% reduction in the size of the procedural representation. The resulting procedural representation was a much more readable and understandable form of the original procedure. This form of the procedural representation would be the form used in the complexity analysis and decomposition phase.

4.2.3 Renaming Map

It was important to keep a two-way mapping between the original CMS-2 symbol names and the new names, so that symbols could be traced from the original CMS-2 code throughout the reverse engineering process. Maintaining this mapping was quite tedious especially as more and more modifications were made at later stages of the reengineering process. This mapping contained not only the name mapping itself but also information about how and why the mapping was made, and recorded the conclusions of any analysis. The reexpression information was also recorded in the renaming map and was used again for later data analysis.

4.2.4 Automation of the Renaming Process

While the choosing of names is a task that requires human insight and understanding, the renaming process involves extensive searching and record keeping that could easily be automated. By automating the location and presentation of references to symbols, the time and effort required to perform renaming would be reduced. The automated generation and maintenance of the renaming map would also be useful. Furthermore, an automated analysis support could be provided for identifying type simplifications.

4.3 Modular Decomposition

Modular decomposition divided existing modules into smaller, more easily handled modules. This division had the advantage of reducing the number of external references per module in the final specification and reducing the complexity of each individual module. The module decomposition was based on separating control blocks in the procedural representation, but other techniques could also be used. The result of the decomposition phase was the separation of the procedural representation into several different modules. Automated analysis tools would be useful for assisting the analysis and mechanics of this process.

4.3.1 Reasons for Modular Decomposition

The RCINSUCC procedure was originally 299 lines of source code, which was equivalent to five pages of text. The size of the procedure made

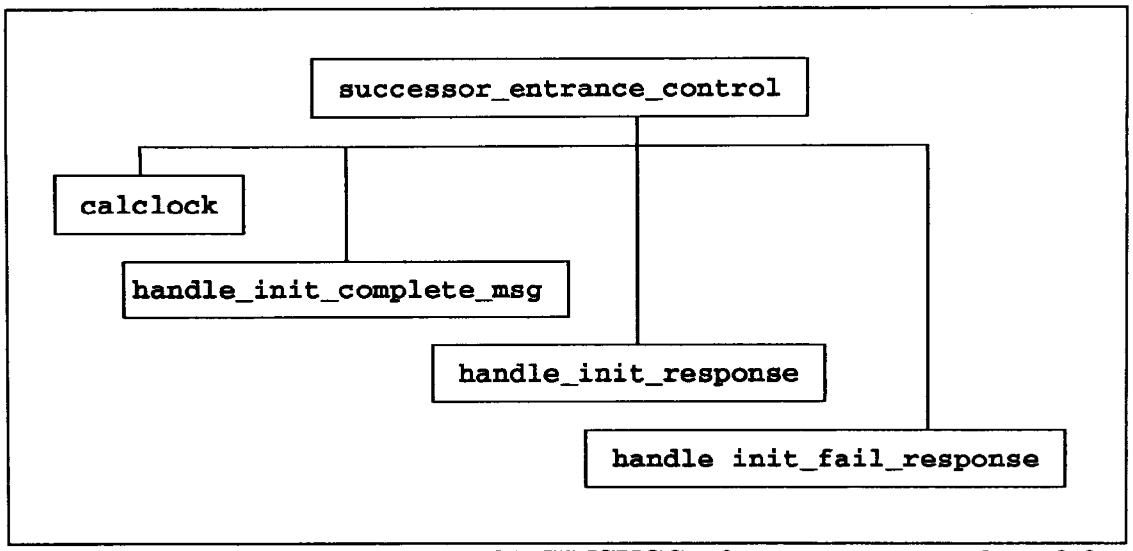

Figure 8. The new structure of RCINSUCC after renaming and modular decomposition.

it difficult to work with and understand. Modular decomposition divided the procedural representation into four additional sub-modules. This reduced the size of the procedural representation of the top level RCINSUCC module to 15 lines, and the representations of sub-modules to 62, 64, 16 and 25 lines each. Figure 8 shows the procedural representation of RCINSUCC after renaming and modular decomposition.

By creating smaller modules, it was easier to reverse engineer the module in parts. Smaller units also resulted in final specifications that were more understandable, because the reduction of unit complexity also made the functionality of each module easier to described. Figure 9 shows procedural representation of the top level submodule, successor_entrance_control. The change in understandability is evident when the procedural representation shown in Figure 9 is compared to the

```
PROCEDURE successor_entrance_control()

BEGIN
  tsd.msg_id = ATESsd.control_word $
  tsd.send_task_id = ATESsd.word_1 $
  IF (tsd.msg_id == tsd_msg_id_INIT) THEN
    handle_init_complete_msg()
  ELSE IF (tsd.msg_id == tsd_msg_id_INIT_RESP) THEN
    handle_init_response_msg()
  ELSE IF (tsd.msg_id == tsd_msg_id_INIT_FAIL_OVERRIDE) THEN
    handle_init_fail_response()
  ELSE
    SSLOGERR(error_str_SUCC_ENT_CONT,tsd.msg_id,NULL)$
  EXEXIT(NULL,NULL) $
END $
```

Figure 9. The procedural representation for the top level module representing RCINSUCC after renaming and decomposition.

original CMS-2 code or even the code's PDL description (see Appendix A). These benefits propagated throughout the reengineering process to the reimplemented C code, which had reduced complexity, making it more maintainable.

Dividing the RCINSUCC procedure into smaller sub-procedures also resulted in fewer external variables being referenced by each module. For example, the original RCINSUCC procedure accessed a total of 37 external data objects. After modular decomposition, the top level module used only one external data object, and the most that were used by any one module was 21. The rest of the external data accesses were contained in the submodules. This decoupling made each module more understandable because the number of external objects that had to be located and understood was reduced.

4.3.2 How Decomposition was Performed

Decisions about how modules should be divided were based on five factors. The size of code blocks was one decomposition criterion. The objective of not allowing overly large modules is a common goal for software. Complexity was another criteria that was used, since reduced complexity made modules more maintainable. Functionality was also a natural way of decomposing modules. While there were no specific rules on what limits to use for each of these criteria, specific guidelines could easily be chosen.

Isolating hardware dependencies was another basis for creating submodules. Separating blocks of code bound to compiler switches allowed them to processed with in later stages of reengineering independently of the main procedural representation. Similarly, direct assembler code could also be isolated by abstracting it into a submodule.

4.3.3 Automation of Modular Decomposition

It would be easy to automate the processes of complexity analysis and decomposition. Modular complexity analysis techniques [19,20] are available that would be directly applicable to this process. Control flow complexity metrics are another means that could be used for decomposing modules, since there is a close relationship between separate control paths and separate functionality. Although automation could improve this process, it is also important that a maintainer's insight be used to assure that meaningful functional decomposition was achieved.

4.4 Data Analysis

Data analysis was one of the most important parts of the reverse engineering process, because it had the greatest impact on improving system understanding. It involved evaluating how data was interrelated, what the data's functionality was, and how data should be reorganized to form abstractions. This process was important for creating specifications that accurately described the data's functionality. More accurate specifications would make the system more understandable and provide better design level encapsulation, which would make new implementations more maintainable.

4.4.1 Objective

The objective of the data analysis phase was to determine information about data relationships that were not immediately evident from the data's declaration or individual references to the data. These relationships included data's functionality, its interaction with other data, its functional composition, and changes that occurred in parallel in different data objects. This information was then used to reorganize the data into abstractions, and determine what operations should be provided for the abstractions. While simplification was the end goal of analysis in the renaming phase, the goal of data analysis phase was improved recomposition of the data.

4.4.2 Analysis Techniques

The data analysis process began in the renaming phase when data declarations and references were examined. This simple analysis allowed data types to be simplified and renamed more appropriately. In the data analysis phase, much more complex analysis was performed. Data relationships and reorganization strategies were determined by analysis of the original data's implementation, use, interaction, and surrounding comments.

Examining how data was implemented provided hints about how what purpose the data served. The data implementation sometimes had to be reconsidered, since the original implementations often were overstated. Mechanisms such as data overlays also provided interesting clues about the data's functionality. Data declarations were also useful for determining data's initialization state.

The references to the data objects were the critical means of determining how data was used. This information was necessary for determining what operators to provide for the reorganized data abstractions. Studying how data was used also provided insight into ways in which complexity could be extracted from the procedural representation and transferred to data abstractions.

Abstraction of functional properties was one way complexity could often be transferred from the procedural representation to data abstractions. For example, Figure 10 shows the declaration and references to an integer variable that was used as a counter. Upon analysis, it was determined that the variables `QCNT1` and `QCNT2` were declared, assigned a

```
*** FILE rcinlocd.inc:
  0423  VRBL QCNT1 I 32 S  ''COUNT OF TASK RESPONSES''$
  0424  VRBL QCNT2 I 32 S  ''COUNT OF TASK RESPONSES''$

*** FILE rcinsucp.inc:
  0296    SET QCNT1 TO 1 $
  0344    SET QCNT1 TO QCNT1+1 $
  0416    SET QCNT2 TO QCNT2+1 $
  0420    SET QCNT2 TO QCNT2 - 1 $
  0422    IF QCNT2 EQ MZNAVTSK
  0454    SET QCNT1 TO QCNT1+1 $
  0455    IF QCNT1 EQ MZINITSK
```

Figure 10. Declaration and references to the variables QCNT1 and QCNT2.

```
counter = data type is init, inc, dec, fetch_value

  Overview
   A counter is a mutable whole number.

  init = proc(c: counter)
       effects Sets the value of c equal to 1.

  inc = proc(c: counter)
       effects Increments the value of c by one.

  dec = proc(c: counter)
       effects Decrements the value of c by one.

  fetch_value = proc(c: counter) returns (int)
       effects Returns the value of c.

end counter
```

Figure 11. Informal specification of the counter data abstraction.

single value, incremented, and decremented. This information lead to the informal specification of a data abstraction to represent these variables, which is shown in Figure 11.

Studying the relationships between different data objects and their uses provided insight into how data objects could be reorganized into more meaningful abstractions. Determining these relationships was a fairly laborious task, because of the numerous types of relationships could exist and the number of possible relationships between objects increased by n! with the number of data objects. It would been easily possible to reduce the number of relationships that had to be checked using various techniques, but because of the limited scope of this research only interrelated data access was analyzed.

Interrelated data access occurred when one data object was referenced or modified in a manner that consistently corresponded to the access or modification of another data object. This often corresponded to the existence of significant functional relationships between data objects. Interrelated data access implied that there was a logical relationship that could be stated about the data, which might be abstracted out of the procedural code and into data abstractions. This relationship could possibly

```
SET QTIMLEFT TO 1024 * (30-(QIODONE-QINISTRT)) $
```

is abstracted to:

update_time_remaining = **proc**(t: timer)

Figure 12. Abstraction of functional information into a data abstraction operator.

be incorporated into a single operator for a data abstraction that performed the function of the previously existing logical relationship between the data.

Figure 10 shows an example of how interrelated data access was determined and abstracted. The variable **QTIMLEFT** is consistently assigned a value based on the variables **QIODONE** and **QINISTRT** throughout the program. This logical relationship leads to the organization of this data into an abstraction that hides the specifics about how the time remaining value is updated.

The analysis of data's conceptual basis provided a means for understanding the designers' intents and the rationale behind the data's original implementation. This information was determined based on the comments accompanying the data declarations, the comments accompanying the data references and the PDL description of the source code. A conceptual basis was critical for making good decisions about how to reorganize the original data into data abstractions.

4.4.3 Integrated Analysis

None of the characterizations of the data was sufficient to develop accurate and meaningful design abstractions for the software. This was because none of the characterizations was complete. For example, data's implementation might have been biased by plans for future changes. This consideration could have led to an implementation that was more complex than it needed to be, making the implementation an inaccurate characterization of the data's current functionality. Coincidental relationships between data could be misinterpreted so that data which was functionally unrelated might be incorrectly grouped in the same abstraction. Comments in the program could also have been misleading, causing the conceptual characterization to not correspond to the code's actual function.

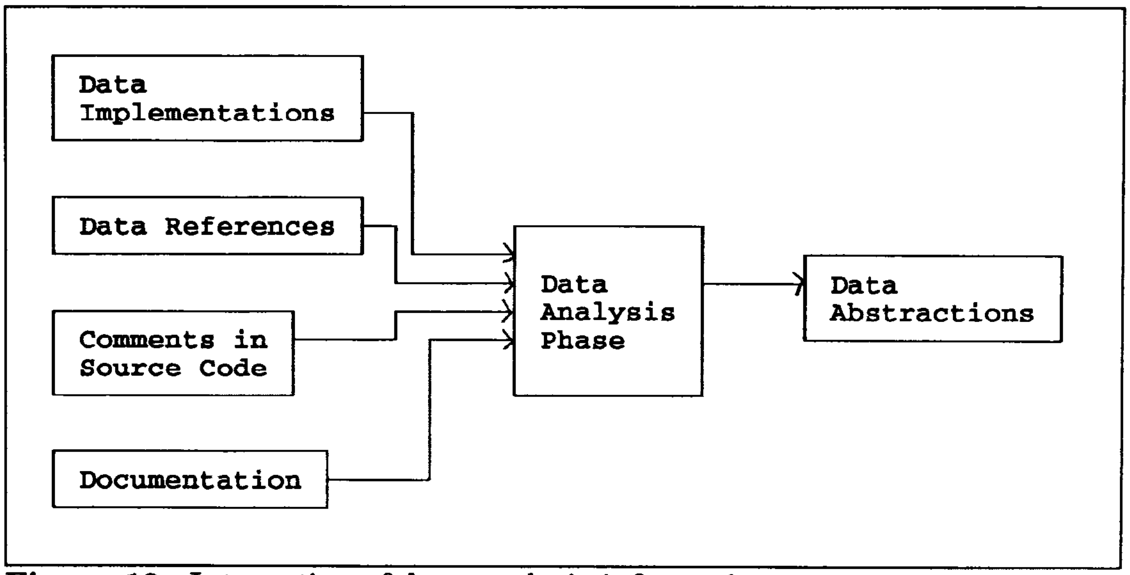

Figure 13. Integration of data analysis information.

Taken together, as shown in Figure 13, it was possible to determine data abstractions that would make the system easier to understand. These abstractions were defined in terms of their operations and informal descriptions of their function. The data abstractions created either mapped to a single data object, multiple data objects, or part of a data object from the original system. Much of the abstraction process was based on human judgement and it is likely that different decisions would lead to different data organizations. Regardless of the variability of the organization, it is most likely that the end result would still be an improvement over the data organization in the original software.

4.4.4 Data-Operator Mapping

As with renaming, it was important to maintain a mapping of the relationship between the renamed CMS-2 data objects and the operators for the new data abstractions. This was necessary so that it would be possible to trace relationships between the original CMS-2 implementation and the reorganized data abstractions. This abstraction map contained information about which data abstraction operators corresponded to original data references. An example of the data mapping for the counter abstraction can be found in Appendix B. This mapping also provided a means of verifying that all of the original CMS-2 data had been accounted for in the reverse engineered data abstractions.

4.4.5 Automation of Data Analysis

The automation of data analysis was critical for the process to be efficient and complete. The process of locating and switching between data declarations and references was a time consuming task. It was further complicated by the data overlay ability that CMS-2 provides, whereby the same data can be referenced by multiple variable names. As with finding references, locating and comparing comments and PDL descriptions of data objects and their references could be improved by providing an effective presentation system. Determining relationships between data was also inefficient, because it was extremely difficult to manually make comparisons of how different data combinations were accessed across the entire program space.

Automation would best be integrated into a decision support system that would provide effective presentation of information and automated analysis of data. An important function of this system would be to automatically display all of the declarations and references to data and improve inspection techniques. Providing statistics about data usage and information about correlations between data objects could also be automatically performed [29]. Automated maintenance of the abstraction mapping would also be useful since it is a simple, but time consuming task to manually keep the map updated.

4.5 Data Recomposition

The goal of data recomposition was to formally define data abstractions that would functionally specify data organization and usage. Data recomposition was achieved through the formulation of Larch specifications. While formulation and composition of LSL specifications was performed manually for this research, automation could be used to assist in constructing the specifications and reduce the time necessary to perform recomposition.

4.5.1 Composition of LSL Specifications

The knowledge gained through the data analysis process was used to form LSL specifications. LSL traits formally specified the operators defined in the data analysis phase. The definition of the corresponding LCL specification was postponed until the forward engineering process.

```
counter: trait

introduces
  COUNTER_start: -> CNTR
  COUNTER_inc: CNTR -> CNTR
  COUNTER_dec: CNTR -> CNTR
  COUNTER_fetch_value: CNTR -> Int

asserts
  CNTR generated by [COUNTER_start, COUNTER_inc, COUNTER_dec]
  CNTR partitioned by [COUNTER_fetch_value]
  forall c : CNTR
    COUNTER_fetch_value(COUNTER_start) == 1;
    COUNTER_inc(COUNTER_dec(c)) == c;
    COUNTER_fetch_value(COUNTER_inc(c)) ==
                                        COUNTER_fetch_value(c) + 1;
    COUNTER_fetch_value(COUNTER_dec(c)) ==
                                        COUNTER_fetch_value(c) - 1
```
Figure 14. LSL specification for a counter.

Automated semantic and syntactic checkers were used to verify the legality of the traits. An example of the end result of data recomposition is shown in Figure 14.

The operators defined in the *introduces* clause of LSL specifications were based on the operators that were determined to be useful in the data analysis phase. Depending upon how the abstraction was formulated, a new operator might perform the function of several operations on the original data. On the other hand, it might take several operators to accomplish what one operation did previously. The former case was more desirable, because it transferred complexity from the procedural representation to the data abstraction, and was more common.

The types that were used in conjunction with the LSL trait operators were also defined. These types including tuples, enumerations, integers and strings. They were chosen based on the types of the data from which the abstraction originated, often with simplified types.

The statement of logical equations for reverse engineered specifications was the final stage of the data recomposition process. The generation of this logic was largely based on the informal specifications created in the data analysis phase and the abstraction mapping from the original CMS-2 code to the reformed data abstractions. Informal specification was useful for providing the specifier with a conceptual notion of the operators' function. The data map was critical in determining the specific logic associated with the operators.

After the LSL specifications were composed it was necessary to verify that they were legal syntactically and semantically. This was accomplished using the LSL checker, which warned of errors that might exist in the specification. The Larch prover might have been used to perform logical checking, but was not used because of time constraints.

4.5.2 Automation of Data Recomposition

The efficiency of the data recomposition phase would be improved by providing an environment in which specifications could be easily composed. Much of the effort required in this phase was in locating data object references in the procedural representation, so that the logic associated with operators could be determined. Automated location and presentation of this information would reduce the time required to perform data recomposition. Automatic generation of LSL skeleton text for the data abstractions would also be useful.

For example, an automated tool could create a skeleton LSL specification based on the information gained in the data analysis phase. It could then prompt the user to define the logic associated with an operator, or a set of operators, by displaying all of the references to the original data that the operator replaced. The tool could determine which references to display using the data-operator map, which was created in the data analysis phase. The tool would perform this process for each of the operators, and then prompt the user to specify any *assumes*, *generated by*, *partitioned by*, *implies converts*, and *exempts* information.

4.6 Procedural Specification

Procedural specification was the stage where the procedural representation was made consistent with the LSL traits. This process required the substitution of operator calls into the procedural representation, and the expression of iteration and iterated blocks in logical form. The procedural specifications were simplified by the information hiding that the data abstractions provided. This conversion could be almost completely automated.

4.6.1 Operator Substitution

The substitution of operator calls for data references in the procedural representation was based on the mapping between the original CMS-2 data and LSL traits. This information was contained in the abstraction mapping. Often the substitutions were direct. Occasionally though, it was necessary to restructure the procedural representation to account for conceptual changes introduced by the data abstractions. This restructuring usually corresponded to removal of iterative or comparative statements.

4.6.2 Expression of Iteration

The substitution of logic and trait operators for iterative constructs in the procedural representation was straightforward. An example of this substitution is shown in Figure 15. A logical *forall* statement is used to

```
VARY QLNDXA WITHIN MZGROUP1 $
     EXQUEUE INPUT MZGROUP1(QLNDXA,TASK),GNINIT,
                        GNMINIT,QM000(0,0) $
END ''VARY'' $
```

maps to...

```
\forall tid : task_id
  (if TASK_is_of_type(tid,task_type_NNAV) then
    EXQUEUE(tid,EXQUEUE_value_INIT,CPCI_msg_INITIALIZATION,
                    INITMSG_get_handle(im'))
  else true)
```

Figure 15. An example of how iteration was respecified.

express the iteration through a set of task ids. A conditional statement and the `TASK_is_of_type` operator are used to state which specific task ids in the set are to be used. This is followed by the statement or group of statements to be performed in each iteration.

4.6.3 Automation of Procedural Specification

Much of the substitution involved in this phase of reverse engineering could be automated since it was accomplished using simple substitution rules. There were more complex cases, though, which required additional work. These cases would be best supported by an environment that would guide the user through the substitution process by making the necessary information available to the user.

4.7 Dependency Analysis and Sequential Specification

Dependency analysis and specification were necessary because the semantics of LSL and LCL did not specify the order in which operations were performed. The sequential execution of the original CMS-2 code implicitly determined an order in which the CMS-2 statements would be executed. To improve the accuracy of the reverse engineered specification, it was necessary to explicitly state these sequential relations in the reverse engineered Larch specifications. Since this process was wholly determined by the original CMS-2 code, its procedural representation, and the data abstractions, the process could be completely automated.

4.7.1 Dependency Analysis

It was necessary to determine which statement(s) in the procedural representation should sequentially follow other statement(s), because of dependencies. The reason for this analysis was that one statement might modify a shared (or global) variable that another statement that followed it might use also. In this case, the second statement would be dependent on the action of the first statement, and must sequentially follow the first statement to perform correctly.

Although the field of dependency analysis has been studied extensively [22,25,26], the method for determining dependency relationships in this research was simple. Dependency analysis was performed at the intramodular level; since all modules had to be called by statements from within other modules, the intermodular case would be accounted for also. In the procedural representation, adjacent operators of different traits were considered independent. External function calls, on the other hand, were always treated as dependent on whatever statements preceded them since it was unclear what data they might access. Similarly, statements that followed an external function call were considered dependent on it.

```
EXSTOPER INPUT RCIN $                                  <- 1

SET MZTSKINI(1,INIT) TO MZINIT $                       <-
SET QCNT1 TO QCNT1+1 $                                 <- 2
SET QRCINVOK(0,REASON) TO 0 $                          <-

EXQUEUE INPUT RCIN,GNSUCC,GNMINIT,QRCINVOK(0,0) $      <- 3
```

Figure 16. Implicit sequential dependency relationships in CMS-2.

An example of the results of dependency analysis is shown in Figure 16. The sample CMS-2 code shows part of a conditional block that has three dependent sequential blocks. The first block is the EXSTOPER function call. The second block contains the next three statements, which reference and modify different data objects. The third block is the EXQUEUE function call. It is important to note that the three assignment statements can be grouped together in the same block because they operate on different data objects.

```
EXSTOPER(task_id_RCIN);

tir' = TIR_set_init(tir^,task_id_IOCD) /\
task_response_count1' =
               COUNTER_inc(task_response_count1^) /\
sir' = SIR_set(sir^,sir_INIT_COMPLETE);

EXQUEUE(task_id_RCIN,EXQUEUE_value_SUCC,
               CPCI_msg_INITIALIZATION,SIR_get_handle(sir^));
```

Figure 17. Sequential specification in the procedural representation.

4.7.2 Sequential Specification

Once the dependency relationships were determined it was necessary to specify them. An example of a sequential specification is shown in Figure 17. This procedural representation corresponds to the CMS-2 source code in Figure 16. There are three dependent statement blocks that are all sequentially dependent on one another. Each of these blocks are specified in groups delimited by a ";" character. The order in which the statements are listed corresponds to the order in which the blocks must be executed. If non-determinism was also a consideration, sequential relationships could have been specified using a modified form of guarded commands [7,16].

4.7.3 Automation of the Analysis and Specification

Determination of sequential dependencies and their subsequent specification is a process that could be performed completely by automated methods. By analyzing modules' procedural representation, dependent code blocks can easily be determined. Sequential specification then only requires containing those blocks using the ";" character and specifying their order.

The dependency determination method used was simple and could be easily improved upon using automated methods. One improvement would be to precisely determine sequential dependency relationships. This could be accomplished by checking external functions to determine what data they reference. This information would allow a more accurate specification of the dependencies of external function calls.

4.8 Generation of LCL Specifications

The final phase of the reverse engineering process was the conversion of the procedural representation to LCL specifications. This was accomplished by generating stub LCL specifications for the LSL traits, *including* and *importing* files created in the reverse engineering process, defining procedural interfaces, removing or commenting out the sequential information, and checking the specifications. The final LCL specification, without the specified sequential information, for the top level procedure that replaced RCINSUCC is shown in Figure 18. The final transformation of the procedural representation to LCL specifications could be automated

```
imports system_functions, task_scheduling_data,
        initialization_msg, date_time_msg,
        ATES_scheduling_data, system_alerts_msg;

uses successor_entrance_control(task_sched_data for TSD,
                                init_msg for IM,
                                sys_alerts_msg for SAM,
                                date_time_msg for DTM,
                                ATES_sched_data for ATESSD);

void successor_entrance_control(void) task_sched_data tsd;
                                      init_msg im;
                                      sys_alerts_msg sam;
                                      date_time_msg dtm;
                                      ATES_sched_data Asd; {
  ensures
    tsd' = TSD_set_msg_id(TSD_set_send_task_id(tsd^,
      ATESSD_get_word_1(Asd^)),ATESSD_get_control_word(Asd^))/\
   (if TSD_is_msg_id(tsd^,tsd_msg_id_INIT) then
      handle_init_complete_msg(tsd^,im^,sam^,dtm^)
    else if TSD_is_msg_id(tsd^,tsd_msg_id_INIT_RESP) then
      handle_init_response_msg(tsd^,im^,sam^)
    else if TSD_is_msg_id(tsd^,
           tsd_msg_id_INIT_FAIL_OVERRIDE) then
      handle_init_fail_response(tsd^, im^, sam^)
    else
      SSLOGERR(error_str_SUCC_ENT_CONT,
              TSD_get_msg_id_handle(tsd^),NULL)) /\
    EXEXIT(NULL,NULL);
  }
```
Figure 18. The LCL specification for successor_entrance_control.

by producing skeleton stubs for the LSL specifications, and by automatically defining the LCL procedural interfaces.

4.8.1 Generating LCL Stubs

It was necessary to generate LCL stubs for the LSL data abstraction traits so that it could be verified that the LCL procedural specifications, which were the result of reverse engineering, were legal specifications and were consistent with the LSL traits. The stubs were generic LCL specifications, which only provided the LCL checker with information about

```
imports task_id;

typedef enum {tsd_msg_id_INIT,
              tsd_msg_id_INIT_RESP,
              tsd_msg_id_INIT_FAIL_OVERRIDE} tsd_msg_id;

abstract type handle;

abstract type task_sched_data;

task_sched_data tsd;

uses task_scheduling_data(task_sched_data for TSD,
                          task_id for TASK_ID,
                          tsd_msg_id for TSD_MSG_ID,
                          handle for HANDLE);
```

Figure 19. The LCL stub for the task_scheduling_data LSL trait.

parameter types and data representations (see Figure 19). Global variables were also declared in the LCL stubs so that the LCL checker could be used.

Parameter types were defined in the LCL specifications based on the parameter types specified in the LSL traits. In the LCL stubs, the data type used for all of the data abstractions, which the LSL traits described, was an *abstract type*. *Abstract types* were chosen, because they did not provide a bias towards how the data should be represented in the new implementation. The decisions about how data abstractions should be reimplemented was left for the forward engineering phase.

4.8.2 Completing the Specifications

To complete the LCL specifications it was necessary to modify the procedural representation so as to explicitly include the necessary traits. These were easily determined by examining the logical statements following the *ensures* clause of the procedural representation. Any trait referenced in that logic had to be included in the LCL specification through either an *imports or a* uses statement.

It was also necessary to fully define the LCL procedural interfaces. This required declaring both direct and global parameters that were passed to the modules defined by the procedural representations. Which parameters to pass explicitly was defined by the original code and the modular decomposition. Global parameters were determined by examining the logic contained in the procedural representation and determining which of the variables used were not passed explicitly to the procedure.

Since sequential specification was not defined in the LCL, the sequential specification determined through reverse engineering was not included in the LCL procedural specifications themselves. Instead, it was intended that the sequential specification, contained in the intermediate procedural representation, would be used in the forward engineering process to determine the sequential constraints that the new implemenation must conform to.

4.8.3 Checking the LCL Specifications

Checking the LCL specifications was the an important and time consuming part of their creation. Using the LCL and LSL checkers it was possible to verify that the specifications were consistent and syntactically legal. They also verified that the same names and types were used consistently in both the LCL procedural specification and the LSL trait specifications. The checking was time consuming because there were a number of inconsistencies and errors that occurred because of the numerous changes that were made. Many of the errors were clerical, some were due to changes that had not been made throughout all of the specifications, and a few errors were mistakes that had been made in the specification process.

It is likely that the number of errors encountered in the specifications would be reduced if the entire reverse engineering process was automated. It was extremely difficult to manually maintain consistency throughout the numerous iterations that were involved in the reverse engineering process. This difficulty resulted in more time being spent correcting inconsistencies when LCL specifications were checked. It is unlikely, though, that the specification mistakes would have been avoided or detected other than by use of the checkers.

4.8.4 Automation of LCL Specification

Much of the work done in the procedural specification phase could easily be automated. In fact, the checking process itself was already automated. None of the other tasks performed, except for error correction, required extensive human comprehension or interaction. At the same time, these tasks were time consuming, making them a prime candidate for automation.

CHAPTER FIVE

FORWARD ENGINEERING

The forward engineering process involved fully defining LCL specifications for the data specifications and then implementing both the procedural and data specifications, which were generated through reverse engineering, C source code. Completing the LCL data specifications required determining how traits should be represented and interfaced to. This information allowed them to be implemented. Implementing the LCL procedural specifications required substitution of the corresponding C code for the LCL logic. While much of this process required complex decision making, automated support could be provided to help implementors choose among possible implementations that were stored in a library, and then automatically perform the necessary substitutions.

5.1 Implementation of the Data Specifications

There were many possible C implementations of the reverse engineered data abstractions. While the specification precisely stated the software's functional requirements, it stated little or nothing about how these functional requirements were to be met. Consequently, it was first necessary to determine how data abstractions would be implemented in C. Once this was determined, it was necessary to complete the LCL specification stubs. Given the LSL and LCL specifications, it was then possible to write the C implementations for the data abstractions.

5.1.1 Determining LSL Trait Implementations

The first step in forward engineering the data specifications was to decide which data specifications would be implemented as data abstractions in C. In creating the LSL specifications, implementation specific information had been abstracted to provide the most flexible and easily understood representation. While these abstractions were useful for better understanding the data's functionality, many would be better represented in a C implementation as simple types. On the other hand, some of the more complex abstractions would be best implemented as abstractions in C to better maintain their properties.

```
flag: trait

  introduces
    new: -> FLAG
    FLAG_set: FLAG -> FLAG
    FLAG_unset: FLAG -> FLAG
    FLAG_check: FLAG -> Bool

  asserts
    generated by [new, set, unset]
    partitioned by [FLAG_check]
    forall f : FLAG
      FLAG_check(FLAG_set(f));
      ~FLAG_check(FLAG_unset(f))
  exempts FLAG_check(new)
```

Figure 20. The LSL specification for a flag.

Two examples of data specifications that were mapped directly to C types were the `flag` specification and the `counter` specification. The `flag` specification, shown in Figure 20, could be implemented as an integer variable in C. There would be little gain in maintainability by implementing it as data abstraction in C. It would just require more time and effort, and reduce program performance.

```
task_initialization_record: trait

  assumes Container(TIR for C), Integer

  introduces
    TIR_clear: TIR -> TIR
    TIR_set_init: TIR, task_id -> TIR
    TIR_is_init: TIR, task_id -> bool
    TIR_get_num_tasks_to_init: TIR -> Int

  asserts
    partitioned by [TIR_is_init,  TIR_get_num_tasks_to_init]
    forall tir : TIR , tid, tid1, tid2 : task_id
      TIR_get_num_tasks_to_init(tir) == 12;
      TIR_is_init(TIR_clear(tir),tid) == false;
      TIR_is_init(TIR_set_init(tir,tid1),tid2) ==
        if tid1 = tid2 then true
        else TIR_is_init(TIR_set_init(tir,tid1))
```

Figure 21. The LSL specification for the task_initialization_record data abstraction.

An example of a data specification that was better implemented as an abstraction would be the `task_initialization_record` specification, shown in Figure 21. This abstraction maps task ids to an initialization state and yields the number of tasks that have to be initialized. It is not specified how the mapping is performed, and it is better kept hidden by an abstraction. Hiding the implementation details allows the representation to be more easily changed and is more easily understood conceptually in the code, since the mapping details are not apparent.

The `task_initialization_record` is a simple example of the type of abstraction that might be better implemented as a data abstraction. The timer data abstraction, shown in Figure 10, is interesting because it abstracts implementation specific information (e.g. computational factors) out of the procedural code. This simplifies maintenance by centralizing where changes are made. In the case of a time-based abstraction, this can be useful since operating system time functions often vary from system to system.

It is less clear for the `counter` specification shown in Figure 14 whether it should be implemented as an abstraction or a C integer type. The operators could be implemented as new C functions for a data abstraction or as existing C functions that operate on integers. The C integer type was chosen because it was felt than no significant loss in maintainability would result, and there would be a significant improvement in performance. Also, the ++ and -- operators in C correspond nicely to the COUNTER_increment and COUNTER_decrement functions specified. It is important to understand that while the use of an integer type would not provide information about the true nature of the data, the LSL/LCL specification would, so that maintainers would have a means of accurately and precisely understanding counter the data type.

An interesting consideration is that it is unlikely that the original system's designers intended for variables used as *counters* to ever be less than zero. This invariant not included in the reverse engineered specifications, because there was information code that justified stating it based on the original system. In the forward engineering process, invariant could easily be added to the specification for a *counter*, as an extension of the specifications, based on intuitive and logical deductions about the *counter* abstraction.

If the invariant:

COUNTER_fetch_value(c) >= 0

had been added to the *counter* data specification, implementing *counters* as C data abstractions would have been more advantageous than as an integer type. A C data abstraction implementation would have employed the data abstraction enforce this invariant, and relieved the procedural code, which used *counters*, from this responsibility.

5.1.2 Completion of LCL Specifications for Data Specifications

After it was determined which data specifications would be implemented as data abstractions, the stub LCL specifications for those specifications were replaced by fully defined LCL interface specifications and then checked. Composing the LCL specifications was an easy task since the LSL specifications contained most of the detail. It primarily involved specifying the names and parameters associated with C procedures used to operate on the implemented data abstractions and how they incorporated the LSL trait operators. The LCL specification for the task_initialization_record is shown in Figure 22.

```
imports task_id;

abstract type task_init_rec;

task_init_rec tir;

uses task_initialization_record(task_init_rec for TIR,
                                   int for Int);

void tir_clear(void) task_init_rec tir; {
  modifies tir;
  ensures tir' = TIR_clear(tir^);
  }

void tir_set_init(task_id tid) task_init_rec tir; {
  modifies tir;
  ensures tir' = TIR_set_init(tir^,tid);
  }

bool tir_is_init(task_id tid) task_init_rec tir; {
  ensures result = TIR_is_init(tir^,tid);
  }

int tir_get_num_tasks_to_init(void) task_init_rec tir; {
  ensures result = TIR_get_num_tasks_to_init(tir^);
  }
```

Figure 22. LCL specification for the task_initialization data abstraction.

5.1.3 Implementing the Data Abstractions in C

Based on the LSL and LCL specifications, the C source code was then written for the abstractions. This process required the implementer to determine a suitable mapping for the abstraction into C and to write C source code that corresponded to the logic contained in the LSL specifications. The implementation of the task_initialization_record is shown in Figure 22.

The advantage offered by implementing the `task_initialization_record` abstraction as a data abstraction is more

```
#include "standard.h"
#include "task_id.h"
#include "task_initialization_record.h"

int task_init_rec[NUM_OF_INIT_TASKS][2] =
           {task_id_DBXC},{task_id_IOCD},{task_id_IO28},
           {task_id_NVCN},{task_id_IOWN},{task_id_IO53},
           {task_id_IOKC},{task_id_DCCN},{task_id_IO19},
           {task_id_SACT},{task_id_WCHP},{task_id_RCMC};

void tir_clear(void)
{
  int i;
  for (i=0; i<NUM_OF_INIT_TASKS; i++)
    task_init_rec[i][INIT_STATUS] = NOT_INIT;
}

void tir_set_init(task_id tid)
{
  int i;
  for (i=0; task_init_rec[i][TASK_ID] != tid; i++);
  task_init_rec[i][INIT_STATUS] = INITIALIZED;
}

int tir_is_init(task_id tid)
{
  int i;
  for (i=0; task_init_rec[i][TASK_ID] != tid; i++);
  if (task_init_rec[i][INIT_STATUS]) return(TRUE);
}

int tir_get_num_tasks_to_init(void)
{
  return(NUM_OF_INIT_TASKS);
}
```

Figure 23. The implementation of task_initialization_record.

apparent when future maintenance requirements are considered. For instance, a logical extension to the program that used this data abstraction might be to check a task's initialization state based on an attribute other than the `task_id`, possibly task priority. By hiding the implementation of the `task_initialization_record`, the implementors would be free to modify the underlying data representation and add new interface procedures without affecting preexisting ones.

5.1.4 The Specification-Implementation Mapping

It was important to maintain a mapping of how the LSL operators were replaced by C implementations. This operator-implementation map made the Larch specifications more useful as documentation for the newly implemented system and made it easier to implement the procedural specifications. The map contains information about the data specifications which had been implemented as data abstractions in C, and those which had been implemented as simple types. For example, the map would show that the LSL operator COUNTER_inc mapped to the ++ function. Similarly, the LSL TIR_is_init operator would map to the tir_is_init function that was implemented.

5.2 Implementation of the Procedural Specifications

Implementing the procedural specifications was a fairly simple task once the data abstractions had been implemented. It involved substituting the mapping from LCL logic to C keywords and constructs, and mapping LSL operator calls to the C implementations of those calls. The C implementation of successor_entrance_control is shown in Figure 24.

5.2.1 Substitution

Mapping from the LCL logic to C keywords was simple but required some judgement in choosing which particular mapping to use. Much of this decision making was stylistic, such as whether a *forall* should be

```
#include "system_functions.h"
#include "task_scheduling_data.h"
#include "initialization_msg.h"
#include "date_time_msg.h"
#include "ATES_scheduling_data.h"
#include "system_alerts_msg.h"

void successor_entrance_control(void)
{
  TSD_set_msg_id(ATESSD_get_control_word());
  TSD_set_send_task_id(ATESSD_get_word_1());
  if (TSD_is_msg_id(tsd_msg_id_INIT))
    handle_init_complete_msg();
  else if (TSD_is_msg_id(tsd_msg_id_INIT_RESP))
    handle_init_response_msg();
  else if (TSD_is_msg_id(tsd_msg_id_INIT_FAIL_OVERRIDE))
    handle_init_fail_response();
  else
      SSLOGERR(error_str_SUCC_ENT_CONT,
                 TSD_get_msg_id_handle(),NULL);
  EXEXIT(NULL,NULL);
}
```

Figure 24. The C implementation of successor_entrance_control.

implemented as a *for*, *while*, or *do* loop or if a logical *if* should be implemented as an *if* or a *switch* statement. Other tasks such as removal of global parameters, conversion from *imports* to *#includes*, and changing "/\" to ";" often required direct substitution.

The LSL operator calls in the LCL specification were substituted with the C implementations for those operators. This substitution was easily accomplished using the operator-implementation map produced during the implementation of the data specifications. Names for direct mapping between the LSL operators and the C data abstraction functions could be substituted directly. Operators that did not have direct mappings into C functions required slightly more complex substitutions.

No multiprocessor implementations were written for this project, so the guarded commands did not have any corresponding C implementations. Instead, the guarded commands were used to determine the order in which the specified dependency blocks would execute in the new C code.

5.3 Automating the Forward Engineering Process

Automation would be particularly useful for maintaining the operator-implementation map, for providing decision support, and in maintaining libraries of reusable C modules. Decision support for choosing possible C data types or possible C data abstractions from an available library of implementations would reduce the time required to implement the specifications and improve the quality of implementations.

It also would be easy to automate the implementation of the procedural specifications. Since most of the substitutions made were based on the operator-implementation mapping (or a simple keyword mapping), it would be easy to create tools that would automatically perform the substitution. Such automation would not only support reengineering efforts; it would also aid in future maintenance activities.

CHAPTER SIX

ANALYSIS OF RESULTS

Analysis of this reengineering project shows that reengineering can provide many of the expected benefits, and that this particular methodology is also useful for addressing systems issues. This analysis presents reengineering examples and an evaluation of the results from which conclusions are drawn.

6.1 Interesting Examples

There are many interesting cases help with understanding how reengineering can improve software quality and maintainability. Three specific examples of reengineered code follow with commentary on the advantages that reengineering provided.

6.1.1 Logical Data Organization

Data reorganization was extremely useful for improving understanding of software as well as improving maintainability by providing more logical grouping of data. The respecification of the `MZTSKINI` table is an example of how data reorganized through reverse engineering. The original CMS-2 declaration of `MZTSKINI` is shown in Figure 25.

`MZTSKINI` was originally used to store the initialization state of certain tasks, and to store information about attributes associated with the tasks. These attributes provided information about the tasks' NAV

```
TABLE MZTSKINI V    3 13  ''TASK INITIALIZATION TABLE'' $
   SUB-TABLE MZGROUP1  2  8  ''NON-NAV DEPENDENT TASKS'' $
   SUB-TABLE MZGROUP2 10  3  ''NAV-DEPENDENT TASKS'' $
   FIELD TASK      I 32 S  0 31
   P DBXC,IOCD,IOWN,IO53,WCHP,
     SACT,IOKC,DCCN,RCMC,NVCN,
     IO19,IO28,IOCD $
   FIELD TASKNM    H  4    1 31
   P H(DBXC),H(IOCD),H(IOWN),H(IO53),H(WCHP),
     H(SACT),H(IOKC),H(DCCN),H(RCMC),H(NVCN),
     H(IO19),H(IO28),H(IOCD) $
   FIELD CRIT      I  8 U  2 23
   P 1,1,7(0),1,3(0) $
   FIELD INIT      I  8 U  2 15 $
END-TABLE MZTSKINI                   $
```
Figure 25. The CMS-2 declaration of the MZTSKINI data table.

dependency and if the tasks were critical. Through data analysis, it was determined that the initialization state for a task was modified during program execution, but the other attributes were not changed.

Based on this knowledge, during data recomposition, the MZTSKINI data was divided into two abstractions (see Figure 26): 1) a *task* abstraction that contained operators capable of determining the attributes associated

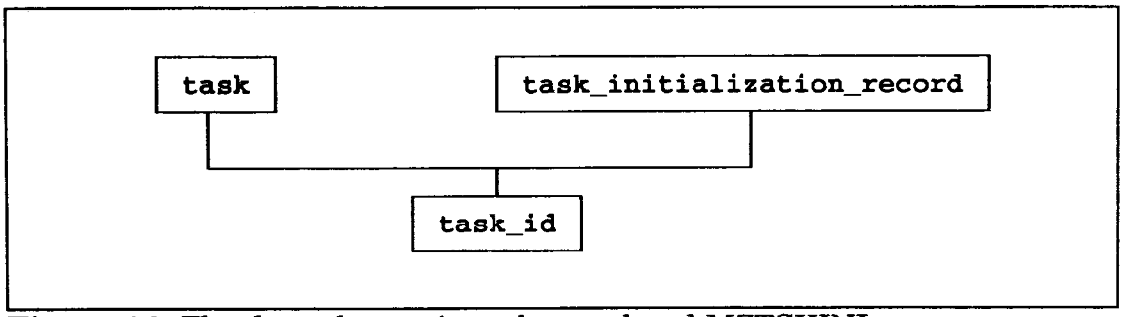

Figure 26. The data abstractions that replaced MZTSKINI.

with a task given its *task_id*; and 2) the *task_initialization_record* (*tir*) abstraction, which was a mutable record of whether a task had been initialized or not. This specification better represented the true nature of the data contained in the MZTSKINI table.

Since the *tir* used information associated with tasks, the task trait was used in its specification. The data division in the new abstractions also had the beneficial side effects of removing implementation details from both the data specification and the procedural specification.

The C implementation of this specification was undetermined. It would be possible, though probably not desirable, to recombine the abstractions into a similar implementation. This implementation choice would be unlikely, because based on the specifications, there was no apparent reason for combining the two data abstractions. It would be more likely that the two would be reimplemented separately so as to provide better modularity. This way, a change in the number of tasks with attributes or the number of attributes associated with a task would not affect the implementation of the *tir*. Similarly, if the *tir* needed to be changed so as to support more than two initialization states, it would not require changing the implementation of the task specification.

6.1.2 Removal of Implementation Details

An example of how implementation dependent aspects of software were abstracted away using Larch specifications is apparent when examining the MZTSKINI example further. In the original CMS-2 implementation, the table MZTSKINI was referenced by an ID number that was contained in a field of the table. To fetch data associated with the ID number, it was first necessary to iterate through the table's ID fields to locate the index associated with the ID. This resulted in additional complexity in the procedures that used **MZTSKINI**.

In the recomposed data abstractions that provided the functionality of the **MZTSKINI** table, the information associated with a *task_id* is referenced directly by the *task_id*. There is no concept of table indices in the specification because it is a purely implementation dependent notion. The result is that the iteration and comparing associated with the table implementation is no longer necessary in the procedural specifications (see

```
VARY QLNDXA WITHIN MZTSKINI $
  IF QDATAPK(0,SENDTSK) EQ MZTSKINI(QLNDXA,TASK)   THEN
    SET MZTSKINI(QLNDXA,INIT) TO MZINIT $
END  ''VARY'' $
```

after data reorganization reduces to

```
tir' = TIR_set_init(tir^,task_id_DBXC) /\
```

Figure 27. Elimination of implementation dependent aspects.

Figure 27).

6.1.3 Removal of Hardware Dependencies

The removal of hardware dependencies was also achieved through data reorganization. One example of this is in the reengineering of the RCINSUCC procedure, where a compiler switch is used to differentiate between two sets of possible values in a task group. The original CMS-2

```
CSWITCH EDC ''RCINSUCP_INC 2370'' $
      BEGIN IOWN,IO53,WCHP,IOED,SACT,DCCN,RCMC
      ''RESPONSE FROM NON-NAV DEPENDENT TASKS''$
END-CSWITCH EDC ''RCINSUCP_INC 2370'' $

CSWITCH KCMX ''RCINSUCP_INC 2370'' $
      BEGIN IOWN,IO53,WCHP,IOKC,SACT,DCCN,RCMC
      ''RESPONSE FROM NON-NAV DEPENDENT TASKS''$
END-CSWITCH KCMX ''RCINSUCP_INC 2370'' $
```

reduces to:

```
else if TASK_is_of_type(TSD_get_send_task_id(tsd^),
                                          task_type_NNAV) then
```

Figure 28. A CMS-2 compiler switch and how it was respecified.

source code containing the compiler switch and the part of the LCL specification corresponding to it are shown in Figure 28. Note that CMS-2's **BEGIN** keyword is equivalent a case statement C.

The compiler switch, which is used to differentiate between implementations for different target platforms, is abstracted out of the procedural specification through the data abstraction process. In the new specification, the logic does not specify at the procedural level which tasks are in which sets. It is only stated that a task must be a member of a particular set to satisfy the condition. The determination of what tasks are in the NNAV set is abstracted to the LSL specification of the *task* trait.

One means of specifying the two possible NNAV task groups would be to specify two different *task* traits. One specification would correspond to the **EDC** switch set and the other would correspond to the **KCMX** switch set. Then, by separating the specifications into separate libraries, depending on the implementation desired, different specification libraries could be used.

The new C implementation could be written in many ways. One possibility would be to create a C implementation of *task* that also used a compiler switch to determine which set of tasks to use. Another alternative might be to create separate C implementations for the *task* trait that would correspond to the two specifications. Like the specifications, the different implementations could be kept in separate libraries that could be chosen at compilation time. It would even be possible, though probably not desirable, to create an implementation that was similar to the original CMS-2 implementation, in which a compiler switch was used in the main procedure to choose the task set to be used.

6.2 Evaluation of Results

The results of this research were useful in providing better understanding of the benefits from reengineering CMS-2 software to C using Larch specifications as an intermediate form. The experience gained in the reengineering process was valuable for developing a complete reengineering methodology. The specifications gained through reengineering provided a better understanding of the system. The C implementation provided a realistic sample of reengineered code. Due to the scope of the project the results were by no means complete, but they were a considerable increase in existing knowledge.

6.2.1 The Reengineering Experience

The results of applying the reengineering process to the RCINSUCC procedure are given in the methodology described in Chapters Four and Five. Applying the process provided insights into how automation could be used. It also helped improve the author's understanding about how to logically represent software. Performing the reengineering was useful for determining the specific challenges that were involved and the areas that need further study.

6.2.2 The Specifications

The goal of providing new documentation and improving system understanding was achieved through the composition of the Larch specifications obtained through reverse engineering. The LSL specifications were useful for providing a better understanding of data use and its implementation independent representation. The LCL procedural specifications were useful for improving understanding of the functionality of the original CMS-2 procedure and the new C implementation. By abstracting implementation details and functional complexity into the LSL specifications, the procedural LCL specifications were made simpler. This made them more easily understandable and made resulting implementations more maintainable.

6.2.3 The New Implementation

The C implementations of the LCL procedural specifications were useful for comparison to the CMS-2 implementation and the Larch specifications. It showed how improved specifications of the CMS-2 code could lead to improved C implementations. The reimplementation process also showed how many different C implementations were possible based on the Larch specifications.

The new C implementation was not executable because many of the data abstractions that the new RCINSUCC implementation used, were not completely reverse engineered, and thus could not be reimplemented. Even if all of the data abstractions had been reimplemented and stubs were used to simulate interaction with other system procedures, running this code would not have provided useful performance information. To gain accurate information about how the new implementation performed, compared to the old implementation, it would be necessary to test the implementation of a much larger part of the system, if not the entire system itself.

6.2.4 Scope Limitations

The extreme interrelation in the ASWCS 116/7 system made it unfeasible to completely reengineer most of its procedures given the scope of this project. For the RCINSUCC procedure it was possible to produce LCL specifications and C code for the procedure. It was not possible to completely reengineer all of the data abstractions that the RCINSUCC procedure used and the other system procedures and functions that it called, because of time constraints. Instead, only some of the data abstractions were fully defined using LSL and LCL specifications, and then reimplemented in C. From these examples it was possible to draw some conclusions about the reengineering process and methodology used.

A smaller example was considered for this project, but was not chosen because it would not have yielded as meaningful results. A smaller self-contained example that could be completely reengineered would have provided a more complete view of the reengineering process, but would not have provided as good a characterization of what it would be like to reengineer most of the system. Smaller examples were also less likely to have as many interesting and diverse cases as larger ones.

CHAPTER SEVEN

CONCLUSION

The research performed was successful in that it accomplished its goals, was a good learning experience, and provided a basis for future research. A reengineering methodology that addressed systems issues was developed and experimented with. There were many lessons learned about the reengineering process, formal specification using Larch, CMS-2 and data abstraction. This research also provided a basis for further research in reengineering, specifically in the area of automation. In drawing these conclusions though, it is important to consider biases that may have been introduced due to the researcher's background.

7.1 Accomplishments

The research goals of developing a reengineering methodology that dealt with systems issues was accomplished by this project. The methodology was developed through the experimentation with a reengineering process, and systems issues were addressed by the reengineering example chosen. An intermediate representation was also used, and important insights were gained about it. Where automation might be applied to improve reengineering process was also considered.

Experimenting with Larch specifications as an intermediate design representation provided great insights into the requirements of design representations and their usefulness. The characteristics on which the Larch family of specification languages was chosen turned out to be important. In particular, flexibility, understandability, automated checking support, and a two-tiered structure were useful in managing the complexities associated with reengineering.

Knowledge of where automation techniques could best be applied was also gained. Manually performing analysis, substitution, and consistency maintenance, provided a strong sense of what processes would best be automated and how a reengineering environment should be constructed. Based on this research, another report is in the process of being written, which discusses the functional requirements of reengineering tools and a support environment [8].

7.2 Lessons Learned

There were many lessons learned as a result of this research: discovery of how complex actual existing systems could be, the need for numerous iterations in the reengineering process, and the need for expertise in the languages and systems used. The complexity of the system used in the experiment was much greater than was initially thought. In particular, the amount of coupling within the system, the complexity of the programming language and operating environment, and the difficulty of understanding the documentation were unexpected.

It was also not clear that reengineering would be such an iterative process. It was expected that data analysis and recomposition could be performed simultaneously. In reality, it was necessary to perform many changes in these phases to obtain the most accurate data abstractions. The need for many of these changes was driven by new data analysis information and the effects of changes that had been made elsewhere in the analysis or recomposition phases.

The need for expertise in the source language, the target language, the intermediate representation, and the existing system were all underestimated. It is critical that the source language and intermediate representation be understood extremely well by the person(s) reverse engineering the system so functionality of the existing software can be correctly interpreted and expressed. Similarly, the person(s) forward engineering the new system must have a firm understanding of the target language and the intermediate representation, so that they can implement the software's representation effectively and accurately.

It is unclear whether it would be beneficial to have the reengineer have expertise with the system being reimplemented. On the one hand, it would be helpful for speeding the reverse engineering process and to better capture the system's intent. On the other hand, the respecification could be easily biased by over-familiarity with the original implementation. It might be best to have a reverse engineer who are less familiar with the system, but who is in close consultation with the previous system maintainers.

7.3 Future Directions

There are several extensions of this research that would be useful for improving this reengineering process. One extension would be to perform more in depth research on techniques that could be applied the different phases of the reverse and forward engineering processes. Developing a set of automated tools and a support environment would make the reengineering process more efficient. Performing a larger scale reengineering project would also be extremely beneficial for addressing topics that this research did not.

Each of the phases in the reengineering processes are research topic in themselves. These topics include source code complexity analysis, data analysis, data mapping, and code generation. The data analysis and data mapping are two primary areas that need to be researched since they are critical to the reverse engineering process. These two areas are necessary for the formulation of an accurate and useful design representation of software.

Developing tools that will aid in analysis and composition is also important for making reengineering an effective and efficient process. Creating a suitable reengineering environment that integrates these tools is another challenge that must be addressed. Much of the need is in developing interfaces that will allow effective manipulation of data and aid in decision support.

While this project was useful for determining basic information, it is necessary to completely reengineer an existing system to better understand the reengineering process. Since this is a fairly large undertaking, it is the next logical step in the course of this research. Reengineering an entire system would provide better insight to some of the system level concerns that were not fully addressed by this research.

Considerable gains in understanding reengineering were made by developing a methodology and reengineering part of a system. Further gains will be made by reengineering an entire system, and then by reengineering multiple systems. But even then, it will not be clear how useful reengineering is until extensive maintenance has been performed on these reengineered systems. This research suggests that reengineering has great potential, but only time and experience will tell.

7.4 Research Biases

In examining this work and its conclusions, it is helpful to know the author and researcher's background, to better understand what biases might have been present. The researcher has a bachelors degree in computer science and has been working in the area of reengineering for slightly over a year. He has four years of programming experience in C, practically no CMS-2 programming experience, little background in formal methods and logic, and no experience with the ASWCS 116/7 system. The author has worked extensively with data abstraction and specification principles.

The reengineering process was aided by the author's understanding of data abstraction and specification. Likewise, it was useful to have a strong background in computer science and a fairly good grasp of the issues involved in reengineering. The C programming experience was helpful in the forward engineering process and in understanding LCL.

Lack of experience with CMS-2, Larch, and the ASWCS 116/7 system was a considerable hinderance in the reengineering process. Not fully understanding CMS-2 and the ASWCS 116/7 system often made it difficult to understand why certain design choices had been made. Lack of experience in writing formal specifications required extensive correction of specifications to improve their accuracy and precision.

Acknowledgements

I would like to thank Professor John Guttag for his supervision, his help with Larch, and his comments and suggestions. I would also like to thank Phil Hwang, Adrien Meskin, and Terri Dumoulin for their comments and suggestions on my thesis, Ali Farsaie and Tamra Moore for their assistance, and Doug Scott for his help with the ASWCS 116/7 system. I also thank Roger Israni and Lynn Boiko for their support and encouragement, and as always, my parents for their love and support.

References

[1] American National Standards Institute, Information Systems - Programming Language - C, X3.159

[2] *USER HANDBOOK FOR CMS-2 COMPILER, REVISION 3*, NAVSEA 0967-LP-598-8020, 1990

[3] Brown, Patrick, "Program Understanding Tools for Systems Software," *Transactions from the Second Annual Systems Reengineering Workshop*, March 1991, pp. 126-135 (to be published)

[4] Bruno, Jeanette M., "An Interactive Approach for Improving Control Flow In An Environment for Code Re-Engineering," *Transactions from the Second Annual Systems Reengineering Workshop*, March 1991, pp. 150-154 (to be published)

[5] Bush, Eric, "The Automatic Restructuring of Cobol," *Proceedings of the Conference on Software Maintenance- 1985*, pp. 35-41

[6] Chikofsky, Elliot J. and Cross, James H. II, "Reverse Engineering and Design Recovery," *IEEE Software*, January 1990, pp. 13-17

[7] Dijkstra, Edsger W., "Guarded Commands, Nondeterminancy and Formal Derivation of Programs," *Communications of the ACM*, Vol. 18, No. 8, August 1975

[8] Duran, Randall E., *A Functional Specification of Reengineering Tools and a Reengineering Environment*, NAVSWC TR 91-390, Naval Surface Warfare Center, Silver Spring, MD (to be published)

[9] Duran, Randall E., "Reengineering CMS-2 Software Using Larch Specifications," *Transactions from the Second Annual Systems Reengineering Workshop*, March 1991, pp. 117-125 (to be published)

[10] Gusaspari, D., Marceau, C., Polak, W., "Formal Verification of Ada Programs," *IEEE Transactions on Software Engineering*, Vol. 16, No. 9, Sept. 1990

[11] Garland, S. J., Guttag, J. V., "An overview of LP, the Larch Prover," *Proc. 3rd Int. Conf. Rewriting Techniques and Applications*, Apr. 1989, pp. 137-151

[12] Garland, S. J., Guttag, Horning, J. J, "Debugging Larch shared

language specifications," *IEEE Trans. Software Eng.*, Vol. 16, No. 9, Sept. 1990, pp. 1044-1057

[13] Guttag, J. V., Horning, J. J., Wing, J. M., "The Larch Family of Specification Languages," *IEEE Software*, Sept. 1985

[14] Guttag, J. V., Horning, J. J., "Introduction to LCL, A Larch/C Interface Language, (to be published)

[15] Guttag, J. V., Horning, J. J., Modet, A., "Report on the Larch Shared Language Version 2.3," *Digital Equipment Corporation Systems Research Center Report 58,* April 1990

[16] Hoare, C. A. R., "Communicating Sequential Processes," *Communications of the ACM*, Vol. 21, No. 8, August 1978

[17] Kernighan, Brian and Ritchie, Dennis, *The C Programming Language*, Prentice-Hall, Inc. Englewood Cliffs, New Jersey, 1978

[18] Liskov, B., Guttag, J. V., *Abstraction and Specification in Program Development*, MIT Press/McGraw-Hill, 1986

[19] McCabe, T. J., "A Complexity Measure," *IEEE Transactions on Software Engineering*, Vol. SE-2 No. 4, December 1976, pp. 308-320

[20] McCabe, T. J., Butler, C. W., "Design Complexity Measurement and Testing," *Communications of the ACM*, Vol. 32, No. 12, December 1989, pp. 1415-1425

[21] Moore, Tamra, Gibson, Katherine, *Reengineering of Navy Computer Systems*, NAVSWC TR 90-216, Naval Surface Warfare Center, Silver Spring, MD (to be published)

[22] Podgurski, A., Clarke, L. A., "A Formal Model of Programming Dependencies and its Implications for Software Testing, Debugging, and Maintenance," *IEEE Transactions on Software Engineering* Vol. 16, No. 9, Sept. 1990, pp. 965-979

[23] Prywes, N., Ge, X., Andrews, S., "Automation of Conversion of Real-Time Software in CMS-2 into Ada," *Transactions from the Second Annual Systems Reengineering Workshop,* March 1991, pp. 81-88, (to be published)

[24] Virrell, A. D., Guttag, J. V., Horning, J. J., Levin, R., "Synchronization Primitives for a Multiprocessor: A Formal Specification," *Digital*

Equipment Corporation Systems Research Center Report 20, August 1987

[25] Wilde, N., Huitt, R., " A Reusable Toolset for Software Dependency Analysis," *Journal of Systems and Software*, Vol. 14, No. 2, Feb. 1991, pp. 97-103

[26] Wilde, N., Nejmeh, B., "Dependency Analysis: An Aid for Software maintenance", SERC-TR-26-F, Software Engineering Research Center, University of Florida, Gainsville, FL, January 1988

[27] Williams, M. H., Ossher, H. L., "Conversion of Unstructured Flow diagrams to Structured Form, *Computer J.*, Vol. 21, No. 2, May 1978, 161-167

[28] Wing, J. M., "Using Larch To Specify Avalon/C++ Objects," *IEEE Transactions on Software Engineering*, Vol. 16. No. 9, Sept. 1990

[29] Zwick, Morris J., "ASWCS 116/7 Automatic Database Design Document Generator," *Transactions from the Second Annual Systems Reengineering Workshop*, March 1991, pp. 155-159 (to be published)

Appendix A: The RCINSUCC Procedure.

```
COMMENT [ BRIEF DESCRIPTION OF PROCEDURE :
          THIS PROCEDURE IS THE SUCCESSOR ENTRANCE CONTROL PROCEDURE
          THAT PROCESSES THE SUCCESSOR ENTRANCE REQUIREMENTS FOR
          INITIALIZATION PROCESSING.
          ] $
COMMENT
          BEGIN RCINSUCP
          . store scheduling packet in local storage
          . DOCASE based on successor data packet
          . CASE 1: initialization complete message            \
          . . DOCASE based on requesting task
          . . CASE 1: responding task is DBXC
          . . . clear DBXC time-out indicator
          . . . IF DBXC initialized properly
          . . . THEN
CSWITCH CALCLOCK
          . . . IF state flag is set to warm
          . . . . THEN
          . . . . . IF there was an operator date time input
          . . . . . THEN
          . . . . . . Set Current Time Clock to temp GMT + Current \
                            Time Clock Delta
          . . . . . . Set GMT to temp GMT
          . . . . . . call SSTIMEAS to convert Current Time Clock in\
                        seconds to DDHHMMss MMssYYss format
          . . . . . . call SSBCDTAG to convert time to BCD
          . . . . . . call SSBCDTAG to convert date to BCD
          . . . . . . format the date returned to sYMD
          . . . . . . call RSETCCC to set the Calendar Clock Card to\
                      the operator input date and time
          . . . . . ELSE
          . . . . . . Set Current Time Clock to output of RASC1985
          . . . . . . Set GMT to the difference between GMT and
                          Current Time Clock delta
          . . . . . ENDIF
          . . . . ELSE
          . . . . . Set Current Time Clock Delta to zero
          . . . . . Set Current Time Clock using temp GMT
          . . . . ENDIF
          . . . . set IOC prior PV to Real Time Clock (requested \
                    by ESCP to be done here as the IOC is tied to the \
                      last time the CTC is updated)
          . . . . set Calibration Factor 1 to remainder of the GMT
                    divided by number of seconds in 1 day - RTC
                    (req by ESCP to maintain a GMT with milliseconds)
          . . . . set Calibration Factor 2 to Calibration Factor 1
          . . . . set previous IOC RTC calue to IOC RTC
END-CSWITCH CALCLOCK
          . . . . set DBXC task initialized in the task       \
                        initialization table
          . . . . set count of initialized tasks to 1
          . . . . IF initialization character was "C"
          . . . . THEN
          . . . . . set system mode word to tactical and the \
                          submodes to w/o
          . . . . . store system mode in the initialiazation \
                          message
          . . . . . call DBSMODE to store system mode in the \
                          database
```

```
. . . . ELSE character was "A", "B" or "W"
. . . . . call DBRMODE to retrieve system mode from\
              the database
. . . . . store system mode in the initialization  \
              message
. . . . ENDIF initialization character
. . . . call SSXTRACT to extract the executive     \
              status data
. . . . reset the restart character in the ATES    \
              table  to "A"
. . . . call EXSYSTRN to save the new              \
              initialization character
. . . . set periodic flag to "IOCD" timeout
. . . . call EXBEGPER to start RCIN initialization   \
               timer at 5 seconds
. . . . queue IOCD at its initalization entrance   \
               with the date/time message
. . . ELSE DBXC did not initialize properly
. . . . call EXDERS with point 29
. . . . queue AMCN with critical, non-override task \
            failure
. . . . queue SD67 with critical, non-override task \
            failure
. . . . queue RCTM with termination message
. . . ENDIF DBXC initialized ok
. . CASE 2: responding task is IOCD
. . . call EXSTOPER to stop periodic scheduling\
            for IOCD timeout
. . . set IOCD task initialized in the task     \
            initialization table
. . . increment count of initialized tasks
. . . set RCIN message buffer to initialization  \
             complete message
. . . queue RCIN at it's successor entrance
. . CASE 3: responding task is RCIN
. . . IF invocation reason is RCIN init complete message
. . . THEN
. . . . call SSGETTIM to get initialization time
. . . . set periodic flag to "all others" responding
. . . . set in progress flag for 30 second timer
. . . . call EXBEGPER to start RCIN initialization   \
               timer at 30 seconds
. . . . DOUNTIL GROUP1 tasks in the initalization    \
               table have been queued
. . . . . queue each task with initialization        \
               messsage at their initialization    \
               entrance
. . . . ENDDO
. . . ELSE (5 second timer has elapsed)
. . . . IF timed out indicator is not cleared
. . . . THEN (DBXC failed to respond)
. . . . . set QPERFLG to critical task failed
. . . . . set failed count to 1
. . . . . call RCINFAIL to process response failure
. . . . ELSE
. . . . . task responded before time elapsed (DO NOTHING)
. . . . ENDIF
. . . ENDIF
. CASE 2: initialization response message from     \
      other tasks
. . DOUNTIL initialization table is searched
. . . IF task id equals task id in data packet
. . . THEN
. . . . IF the response indicates a successful        \
            intialization
. . . . THEN
. . . . . set the active field for the task
. . . . . DOCASE base on responding task ID
. . . . . CASE 1 response from NVCN
. . . . . . IF periodic is in NVCN timeout mode
. . . . . . THEN
. . . . . . . call EXSTOPER stop periodic
. . . . . . ENDIF
. . . . . . set periodic flag to NAV-dependent IOs
. . . . . . start periodic for 2 second timeout
. . . . . . queue NAV dependent I/O tasks with          \
             initialization message
. . . . . CASE 2 response from NAV-dependent task
. . . . . . increment NAV dependent task counter
. . . . . . IF task is critical IOCD
. . . . . . THEN
. . . . . . .  decrement NAV dependent task counter
. . . . . . ENDID
. . . . . . IF NAV dependent task counter indicates all    \
                tasks answered
. . . . . . THEN
. . . . . . . stop periodic processing for two second      \
                timeout
. . . . . . . IF 30 second timer has not expired
. . . . . . . THEN
. . . . . . . . set periodic flag back to "all others"
. . . . . . . . get current time
. . . . . . . . calculate time left for 30 sec timer
. . . . . . . . call EXBEGPER to restart periodic
. . . . . . . ENDIF
. . . . . . ENDIF
. . . . . CASE 3 response from all non-NAV dependent \
              tasks
. . . . . . increment task initialization counter
. . . . . . IF the task initialization counter indicates    \
                 all tasks initialized
. . . . . . THEN
. . . . . . . call EXDERS with point 29
. . . . . . . IF two second timer is in progress
. . . . . . . THEN
. . . . . . . . clear 30 sec. in progress flag (QINPROG)
. . . . . . . ELSE
. . . . . . . . call EXSTOPER to stop the periodic        \
                   processing for RCIN
. . . . . . . ENDIF
. . . . . . . IF initialization character is "A"
. . . . . . . THEN
. . . . . . . . queue AMCN with auto reload complete        \
                  message
. . . . . . . ENDIF initialization character
. . . . . . . queue SACT at its successor entrance with     \
                 initialization complete
. . . . . . . queue SD67 at its successor entrance with     \
                 initialization complete
. . . . . . . call SSTYPEIT with initialization complete
. . . . . . ENDIF all tasks initialized
. . . . . ENDCASE
. . . . . ENDIF
. . . . ENDIF successful initialization
. . . ENDIF proper task id
. . ENDDO search initialization table
```

```
          . CASE 3: SD67 "NO" or "GO"
          . . IF "NO" is received from SD
          . . THEN
          . . . set termination flag in the task data base
          . . . queue RCTM with termination request,          \
                      non-critical task failure
          . . ELSE "GO" received from SD
          . . . IF initialization character is "A"
          . . . THEN
          . . . . queue AMCN with auto reload complete        \
                         message
          . . . ENDIF initialization character
          . . . queue SACT at its successor entrance with     \
                      initialization complete
          . . . queue SD67 at its successor entrance with     \
                      initialization complete
          . . . call SSTYPEIT with initialization complete
          . . ENDIF answer from SD
          . CASE 4: "software execption"
          . . call SSLOGERR to issue software exception       \
                   message
          . ENDCASE
          . call EXEXIT to exit successor entrance
          END RCINSUCC
          ] $
COMMENT ((EJECT $
(EXTDEF)  PROCEDURE RCINSUCC $
          LOC-INDEX QLNDXA,QLNDXB           ''LOCAL INDEX'' $
          SET QDATAPK(0,0) TO GNSCHPKT(QDUM,ASWSCW) $
          SET QDATAPK(0,1) TO GNSCHPKT(QDUM,SDW) $
          FOR QDATAPK(0,MSGID)
          ELSE                        ''CASE 4''
             BEGIN $
             SSLOGERR INPUT H(RCINSUCC),QDATAPK(0,MSGID),0 $
             END                      ''END CASE 4'' $
             BEGIN GNMINIT            ''CASE 1 - COMPLETE RCIN INIT.
                                      REQUEST'' $
             FOR QDATAPK(0,SENDTSK)
             ELSE
                BEGIN ''CASE 4 - EXCEPTION CASE'' $
                SSLOGERR INPUT H(RCINSUCC),
                              QDATAPK(0,SENDTSK),0 $
                END $

                BEGIN DBXC ''CASE 1 - RESPONSE TASK IS DBXC'' $
                SET QTIMEOUT TO 0 $
                IF GNSCHPKT(QDUM,INITSTAT) EQ QOK
                THEN                        ''DBM INITIALIZATION OK''
                   BEGIN $
CSWITCH CALCLOCK $
                   IF QM000(0,RFLAG) EQ 0
                   THEN
                     BEGIN $
                     IF QOPENTRY EQ 1
                     THEN
                       BEGIN $
                         SET QSETCCC TO 0 $
                         SET QTTIME6(0,QCTC) TO QGMTTEMP $
                         SET QTTIME6(0,QDATGMT) TO QGMTTEMP $
                         SET QTTIME6(0,QDELTA) TO 0 $
                         SET QTTIME6(0,QDELTAMS) TO 0 $
                         SSTIMEAS INPUT QTTIME6(0,QCTC)
                             OUTPUT QOUT1,QOUT2 $
                         RTAGBCD INPUT QOUT1 OUTPUT QTEMTIME $
                         SET CHAR(0,2)(QSETCCC(0,BCDTIME)) TO
                                               CHAR(2,2)(QTEMTIME) $
                         RTAGBCD INPUT QOUT2 OUTPUT QTEMDATE $
                         ''FORMAT TO sYMD TO SET CCC''$
                         SET CHAR(1,3)(QSETCCC(0,BCDDATE)) TO
                                                CHAR(1,3)(QTEMDATE) $
                         SET EXB1 TO CORAD(QSETCCC) $
                         EXEC 60 $
                       END $
                     ELSE
                       BEGIN $
                         SET QTTIME6(0,QCTC) TO QGMTTEMP $
                         SET QTTIME6(0,QDATGMT) TO QGMTTEMP $
                         SET QTTIME6(0,QDELTA) TO 0 $
                         SET QTTIME6(0,QDELTAMS) TO 0 $
                       END $
                     END $
                   ELSE
                     BEGIN $
                       SET QTTIME6(0,QDELTA) TO 0 $
                       SET QTTIME6(0,QDATGMT) TO QGMTTEMP $
                       SET QTTIME6(0,QCTC) TO QGMTTEMP $
                     END $
                   ''ENDIF'' $
                   SET QTTIME6(0,QPIOC) TO QRTC $
                   SET QCLKGMT6(0,CALFAC1) TO REM(QGMTTEMP/86400) -
                        QRTCMS/1000 $
                   SET QCLKGMT6(0,CALFAC2) TO QCLKGMT6(0,CALFAC1) $
                   SET QCLKGMT6(0,PIOCRTC) TO QRTC $
END-CSWITCH CALCLOCK $
                   SET MZTSKINI(0,INIT) TO MZINIT $
                   SET QCNT1 TO 1 $
                   IF QM000(0,RFLAG) EQ QCOLD
                   THEN
                      BEGIN                 ''COLD START'' $
                      SET QM000(0,SMODE) TO GNNORMAL $
                      SET QM000(0,SUBMODE) TO GNNORMAL $
                      SET QMODE TO GNNORMAL ''SET MODE TO TACT'' $
                      SET QSMODE TO QNONE ''NO SUBMODE FOR TACT'' $
                      DBSMODE INPUT QMODE,QSMODE OUTPUT QSTAT $
                      END                   ''COLD START'' $
                   ELSE
                      BEGIN                 ''WARM START'' $
                      DBRMODE OUTPUT QSTAT,QMODE,QSMODE,QENG $
                      SET QM000(0,SMODE) TO QMODE ''CURRENT SYST
                                                    MODE'' $
                      SET QM000(0,SUBMODE) TO QSMODEO
                                                 ''AND SUBMODE'' $
                      END                   ''WARM START'' $
                   SSXTRACT INPUT QXSTAT ''GET THE EXEC STAT DATA'' $
                   SET QATESPK(0,ICHAR) TO H(A) ''AUTO WARM RELOAD'' $
                   EXSYSTRN INPUT QSAVE ''SAVE THE NEW INIT CHAR'' $
      COMMENT   MOVE DATA FROM LOCAL TABLE TO CONTEXT AREA TABLE $
                   SET QRCTIME6 TO QMMANTIM $
                   SET QPERFLG TO QIOCD $
                   EXBEGPER INPUT 5120,RCIN $
                   EXQUEUE INPUT IOCD,GNINIT,GNMINIT,QM000(0,0) $
                   END                      ''DBM INITIALIZATION OK'' $
                ELSE                        ''DBM INITIALIZATION FAILED''
                   BEGIN $
                   SET QMTERMRQ(0,REQ) TO QRCIN $
                   SET QMTERMRQ(0,FAILTYPE) TO QDBM $
```

```
            SET QAMCNBUF(0,ALERTID) TO QFAIL $
            EXDERS INPUT 29,0,0 ''SYSTEM DERS POINT'' $
            EXQUEUE INPUT AMCN,GNSUCC,GNALERT,CORAD(QAMCNBUF) $
            EXQUEUE INPUT SD67,GNSUCC,MZTERMRQ,QMTERMRQ(0,0) $
            EXQUEUE INPUT RCTM,GNSUCC,MZTERMRQ,QMTERMRQ(0,0) $
            END                ''DBM INITIALIZATION FAILED'' $
         ''ENDIF'' $
         EXDERS INPUT 601,0,0  ''SYSTEM DERS POINT'' $
         END ''CASE 1 - RESPONSE TASK IS DBXC'' $
         BEGIN IOCD  ''CASE 2 - RESPONSE FROM IOCD'' $
         EXSTOPER INPUT RCIN $
         SET MZTSKINI(1,INIT) TO MZINIT $
         SET QCNT1 TO QCNT1+1 $
         SET QRCINVOK(0,REASON) TO 0 $
         EXQUEUE INPUT RCIN,GNSUCC,GNMINIT,QRCINVOK(0,0) $
         END ''CASE 2 - RESPONSE FROM IOCD'' $
         BEGIN RCIN ''CASE 3 - RESPONSE FROM RCIN'' $
         IF QDATAPK(0,RCREASON) EQ 0
         THEN
            BEGIN ''RCIN initialization message'' $
            SSGETTIM OUTPUT QINISTRT $
            SET QPERFLG TO QOTHERS $
            SET QINPROG TO 1 $
            EXBEGPER INPUT 30720,RCIN $
            VARY QLNDXA WITHIN MZGROUP1 $
            EXQUEUE INPUT MZGROUP1(QLNDXA,TASK),GNINIT,
                     GNMINIT,QM000(0,0) $
            END ''VARY'' $
            END ''RCIN initialization message'' $
         ELSE
            BEGIN ''DBXC 5 SEC TIMER ELAPSED'' $
            IF QTIMEOUT
            THEN
               BEGIN ''DBXC FAILED TO RESPOND'' $
               SET QPERFLG TO QDBXC $
               SET QFLCNT TO 1 $
               RCINFAIL $
               END ''DBXC FAILED TO RESPOND'' $
            END ''DBXC 5 SEC TIMER ELAPSED'' $
         ''ENDIF'' $
         END  ''CASE 3 - RESPONSE FROM RCIN'' $
      END   ''FOR'' $
      END   ''CASE 1 - RCIN INIT. COMPLETION'' $
      BEGIN GNMINITR         ''CASE 2 - INIT. RESPONSE
                             MESSAGE'' $
      VARY QLNDXA WITHIN MZTSKINI $
         IF QDATAPK(0,SENDTSK) EQ MZTSKINI(QLNDXA,TASK)
         THEN
            BEGIN ''SENDING TASK ID CHECK'' $
            IF QDATAPK(0,INITSTAT) EQ QOKINIT
            THEN
               BEGIN  ''SUCCESSFUL INITIALIZATION'' $
               SET MZTSKINI(QLNDXA,INIT) TO MZINIT $
               FOR QDATAPK(0,SENDTSK)
               ELSE                     ''CASE 4''
                  BEGIN $
                  SSLOGERR INPUT H(RCINSUCC),
                                 QDATAPK(0,MSGID),0 $
                  END                   ''END CASE 4'' $
                  BEGIN NVCN  ''RESPONSE FROM NVCN'' $
                  IF QPERFLG EQ QNVCN
                  THEN
                     BEGIN  ''NVCN TIMEOUT ACTIVE'' $
```

```
                                 EXSTOPER INPUT RCIN $
                                 END    ''NNCN TIMEOUT ACTIVE'' $
                              IF QPERFLG EQ QNVCNB
                              THEN
                                 BEGIN  ''NAV DATA AVAILABLE'' $
                                 SET QPERFLG TO QNAVIOS $
                                 EXBEGPER INPUT 2048, RCIN $
                                 VARY QLNDXB WITHIN MZGROUP2 $
                                    EXQUEUE INPUT MZGROUP2(QLNDXB,TASK),
                                    GNINIT,GNMINIT,QM000(0,0) $
                                 END ''VARY'' $
                                 END     ''NAV DATA AVAILABLE'' $
                              ELSE
                                 SET QPERFLG TO QNVCNB $
                              END ''RESPONSE FROM NVCN'' $
                              BEGIN IO19,IO28,IOCD ''NAV DEPENDENT I/Os'' $
                              SET QCNT2 TO QCNT2+1 $
                              IF MZTSKINI(QLNDXA,TASK) EQ IOCD AND
                                 MZTSKINI(QLNDXA,CRIT) EQ 1
                              THEN
                                 SET QCNT2 TO QCNT2 - 1 $
                              ''ENDIF''
                              IF QCNT2 EQ MZNAVTSK
                              THEN
                                 BEGIN ''ALL NAV-DEPENDENT I/Os
                                       RESPONDED'' $
                                 EXQUEUE INPUT SACT,GNSUCC,MZINITNV,
                                             QM000(0,0) $
                                 EXSTOPER INPUT RCIN $
                                 IF QINPROG
                                 THEN
                                    BEGIN  ''RESTART PERIODIC'' $
                                    SET QPERFLG TO QOTHERS $
                                    SSGETTIM OUTPUT QIODONE $
                                    SET QTIMLEFT TO
                                        1024*(30-(QIODONE-QINISTRT)) $
                                    IF QTIMLEFT GT 0
                                    THEN
                                       EXBEGPER INPUT QTIMLEFT, RCIN $
                                    ELSE
                                       EXBEGPER INPUT 1, RCIN $
                                    ''ENDIF'' $
                                    END ''RESTART PERIODIC'' $
                                 END ''ALL NAV-DEPENDENT I/Os RESPONDED'' $
                              END  ''NAV-DEPENDENT I/Os'' $
CSWITCH EDC ''RCINSUCP_INC 2370'' $
                              BEGIN IOWN,IO53,WCHP,IOED,SACT,DCCN,RCMC
                              ''RESPONSE FROM NON-NAV DEPENDENT TASKS'' $
END-CSWITCH EDC ''RCINSUCP_INC 2370'' $
CSWITCH KCMX ''RCINSUCP_INC 2370'' $
                              BEGIN IOWN,IO53,WCHP,IOKC,SACT,DCCN,RCMC
                              ''RESPONSE FROM NON-NAV DEPENDENT TASKS'' $
END-CSWITCH KCMX ''RCINSUCP_INC 2370'' $
                              SET QCNT1 TO QCNT1+1 $
                              IF QCNT1 EQ MZINITSK
                              THEN
                                 BEGIN  ''TASK INIT. COUNT COMPLETED'' $
                                 EXSTOPER INPUT QTASK $
                                 IF MZTSKINI(9,INIT) EQ MZNOINIT
                                 THEN
                                    BEGIN ''RESTART TIMEOUT FOR NVCN'' $
                                    SSGETTIM OUTPUT QIODONE $
                                    SET QPERFLG TO QNVCN $
```

```
                    SET QTIMLEFT TO
                        1024*(30-(QIODONE-QINISTRT)) $
                    EXBEGPER INPUT QTIMLEFT, RCIN $
                    END ''RESTART PERIODIC FOR NVCN'' $
                 EXDERS INPUT 29,0,0 ''SYSTEM DERS POINT''$
                 SET QINPROG TO 0 $
                 IF QM000(0,INITCH) EQ H(A)
                 THEN
                    BEGIN $
                    SET QAMCNBUF(0,ALERTID) TO QREL $
                    EXQUEUE INPUT AMCN, GNSUCC, GNALERT,
                      CORAD(QAMCNBUF) $
                    END $
                 EXQUEUE INPUT SACT,GNSUCC,MZINITOK,
                               QM000(0,0) $
                 EXQUEUE INPUT SD67,GNSUCC,MZINITOK,
                               QM000(0,0) $
                 SSTYPEIT INPUT CORAD(QOCD(QICM,TEXT)),7 $
                 END       ''TASK INIT. COUNT CHECK'' $
              END ''RESPONSE FROM NON-NAV DEPENDENT
                    TASKS'' $
           END ''FOR'' $
           END  ''SUCCESSFUL INITIALIZATION'' $
        END ''SENDING TASK ID CHECK'' $
  END  ''VARY'' $
  END  ''END CASE 2 - INITIALIZATION RESPONSE MESSAGE'' $
  BEGIN MZIFRMID       ''CASE 3 - INIT. FAIL RESPONSE'' $
  IF QDATAPK(0,OFAIL) EQ QNOGO
  THEN                 ''NO RECEIVED FROM SD67''
     BEGIN $
     SET QTMFG TO 1 $
     SET QMTERMRQ(0,REQ) TO QRCIN $
     SET QMTERMRQ(0,FAILTYPE) TO QNCRT $
     EXQUEUE INPUT RCTM,GNSUCC,MZTERMRQ,QMTERMRQ(0,0) $
     END               ''NO RECEIVED FROM SD67'' $
  ELSE                 ''GO RECEIVED FROM SD67''
     BEGIN $
     IF QM000(0,INITCH) EQ H(A)
     THEN
     BEGIN $
        SET QAMCNBUF(0,ALERTID) TO QREL $
        EXQUEUE INPUT AMCN, GNSUCC, GNALERT,
        CORAD(QAMCNBUF) $
     END $
     EXQUEUE INPUT SACT,GNSUCC,MZINITOK,QM000(0,0) $
     EXQUEUE INPUT SD67,GNSUCC,MZINITOK,QM000(0,0) $
     SSTYPEIT INPUT CORAD(QOCD(QICM,TEXT)),7 $
  END                  ''GO RECEIVED FROM SD67'' $
  END                  ''END CASE - 3 INIT. FAILURE
                       RESPONSE'' $
END                    ''END FOR'' $
EXEXIT INPUT 0,0 $
END-PROC RCINSUCC  $
```

Appendix B: The Data-Operator Map for the *counter* Abstraction

```
*** FILE rcinlocd.inc:
  0423  VRBL QCNT1 I 32 S  ''COUNT OF TASK RESPONSES''$

  maps to counter

  0424  VRBL QCNT2 I 32 S  ''COUNT OF TASK RESPONSES''$

  maps to counter

*** FILE rcinsucp.inc:
   0296     SET QCNT1 TO 1 $

  maps to COUNTER_init

   0344     SET QCNT1 TO QCNT1+1 $

  maps to COUNTER_inc

   0416     SET QCNT2 TO QCNT2+1 $

  maps to COUNTER_inc

   0420     SET QCNT2 TO QCNT2 - 1 $

  maps to COUNTER_dec

   0422     IF QCNT2 EQ MZNAVTSK

  maps to COUNTER_fetch_value

   0454     SET QCNT1 TO QCNT1+1 $

  maps to COUNTER_inc

   0455     IF QCNT1 EQ MZINITSK

  maps to COUNTER_fetch_value
```