Evolution of Batch-Oriented COBOL Systems into Object-Oriented Systems through Unified Modelling Language

Richard Charles Millham

A thesis submitted in partial fulfilment of the requirements for the degree of

Doctor of Philosophy

De Montfort University

February, 2005

Abstract

Throughout the world, there are many legacy systems that fulfil critical business functions but often require new functionality to comply with new business rules or require redeployment to another platform. Legacy systems vary tremendously in size, functionality, type (such as batch-oriented or real-time), programming language source code, and many other factors. Furthermore, many of these legacy systems have missing or obsolete documentation which makes it difficult for developers to re-develop the system to meet any new functionality. Moreover, the high cost of whole scale redevelopment and high switchover costs preclude any replacement systems for these legacy systems. Reengineering is often proposed as a solution to this dilemma of high re-development and switchover costs.

However, reengineering a legacy system often entails restructuring and re-documenting a system. Once these restructuring and re-documentation processes have been completed, the developers are better able to redevelop the parts of the systems that are required to meet any new functionality. This thesis introduces a number of methods to restructure a procedurally-structured, batch-oriented COBOL system into an object-oriented, event-driven system through the use of an intermediate mathematical language, the Wide Spectrum Language (WSL), using system source code as the only documentation artefact. This restructuring process is accomplished through the application of several algorithms of object identification, independent task evaluation, and event identification that are provided in the thesis. Once these transformations are complete, method(s) are specified to extract a series of UML diagrams from this code in order to provide documentation of this system. This thesis outlines which of the UML diagrams, as specified in the UML Specifications version 1.5, can be extracted using the specified methods and under what conditions this extraction, using system source code only, can occur in a batch-oriented system. These UML diagrams are first expressed through a WSL-UML notation; a notation which follows the semantics and structure of UML Specifications version 1.5 in order to ensure compatibility with UML but is written as an extension of WSL in order to enable WSL to represent abstract modelling concepts and diagrams. This WSL-UML notation is then imported into a visual UML diagramming tool for the generation of UML diagrams to represent this system.

The variety of legacy systems precludes any universal approach to reengineering. Even if a legacy system shares a common programming language, such as COBOL, the large number of COBOL constructs and the huge number of possible dialects prevents any universal translator of the original program code to another. It is hoped that by focusing on one particular type of legacy system with constraints, in this case a batch-oriented COBOL system with its source code its only surviving artefact, and by providing validated algorithms to restructure and re-document these legacy systems in the Unified Modelling Language, an industry system modelling standard, and by determining which of these Unified Modelling Language can be extracted practically from such a system, some of the parameters and uncertainties, such as program understanding of an undocumented system, in reengineering this type of system can be reduced.

Acknowledgments

The author wishes to thank principally his first supervisor, Hongji Yang, for all his help, advice, and suggestions during this research process. Thanks must also go to my second supervisors, Professor Hussein Zedan and Mr. Steve McRobb, for all the help and advice that they provided along the way. Additionally, I would like to thank my local supervisor, Dave Shellenberg, for his helpful comments and suggestions as well as lending his expertise with legacy systems.

I would like to dedicate this thesis to the memory of Jan Kwiatkowski, a former research student of Hongji Yang, whose arduous task in formulating a tool to translate common COBOL constructs into WSL was tragically cut short by his death in a car accident several years ago.

Contents

List of Figures

List of Tables

Chapter 1: Introduction

1.1 Purpose of Research and Overview of Problem

Legacy systems can be defined as long-term mission critical software systems that contain comprehensive business knowledge and that constitute large assets for organizations. However, due to maintenance activities, their quality and operational life are constantly deteriorating (Zhou, 2003).

These legacy systems, in many cases, were designed in an ad-hoc manner and were designed as stand-alone systems. As time passed and business needs changed, the need to integrate these disparate systems together and remodel them to accommodate a multi-tiered, Web-based platform became apparent. The move of legacy systems from a sequential-driven, procedural structured to an event-driven, component-based architecture can be achieved by transformations of the original system to an object-oriented, event-driven system. Object orientation, because it encapsulates methods and their variables into modules, is well-suited to a multi-tiered Web-based architecture where pieces of software must be defined in encapsulated modules with cleanly-defined interfaces. This particular legacy system is procedural-driven and is batch-oriented; a move to a Web-based architecture requires a real-time, event-driven response rather than a procedural invocation (Newcomb, 2001, Ulrich, 2000; Zhou, 2003).

Object orientation promises other advantages as well. Because object orientation has encapsulated modules of software with cleanly-defined interfaces, object-oriented software systems, and hence their objects, from different systems can be more easily integrated than if this software is structured into procedures with global variables. Furthermore, object orientation has lower maintenance costs than procedural software (Rumbaugh, 1991;Fergen, 1994; Meyer, 1987).

In order to integrate disparate systems, developers must fully understand the systems being integrated. They must not only understand the software structure of the system but they must also be able to follow the data and control flow and the execution of events (Ulrich, 2000, Newcomb 2001). Documentation detailing this information for most legacy systems is either missing or out-of-date. Consequently, any reengineering efforts must first have some facility to generate documentation regarding the structure and dynamics of this system, usually based on the system code alone (Ball, 1996).

Another problem of legacy systems is that legacy systems are written in obsolete languages such as COBOL which are difficult to port to other platforms. The number of COBOL-trained programmers is shrinking. Furthermore, COBOL is generally not suitable for developing programs for Web-based architectures. Consequently, there is a need to translate the system into a language more suitable for Web-based architectures, such as C++ (Newcomb, 2001).

Because legacy systems fulfil a critical business need yet the high cost of development prohibits developing replacement systems, one solution to this dilemma is to reengineer these legacy systems to fit new architectures and business requirements (Newcomb, 2001).

This thesis contains a discussion as to how to address the restructuring and re-documentation of the problems in reverse engineering, in regards to batch-oriented systems, based on the following observations:

1) Existing documentation is often incomplete; there is a need to provide documentation of this legacy system in order to understand this system. Understanding the system is necessary in order to change it to meet new business needs whether the end product is new development or reengineering of the existing application.

2) Formal methods are able to provide a theoretical basis for integrating transformation techniques in constructing a practical software reverse engineering tool.

3) Object orientation, which encapsulates software by grouping data and operations that manipulate them into modules with cleanly-defined interfaces, provides the ability to more easily integrate objects from disparate systems together than if these disparate systems were structured procedurally.

1.1.1 Motivation

1.1.1.1 Business Case

The motivation for this research came as a result of a gap that was recognised between an unfulfilled business situation and a gap in tools and methodologies to resolve it.

A telecommunications company had, as its set of core systems, a number of standalone mainframe legacy systems written in COBOL. Aside from many incremental changes over the years and the removal of the original user interface, to be replaced by Visual Basic screen scrapers in the 1980's, these legacy systems remained because they fulfilled a crucial business need. Attempts to replace these systems with newly-developed systems or with similar systems purchased from other telecommunications companies often ended in failure, for a number of good reasons. Some of these reasons included different legal regulations, high cost of new development, and high switchover times. Replacement systems, therefore, were not an option (Telecommunications Employee C, 1999).

During this time, the telecommunication company's systems had a large number of new business needs and rules to implement within their core systems. Data services, mobile services, and organisational changes due to mergers required that these core systems be updated to reflect these new business needs and rules (Telecommunications Employee C, 1999).

These difficulties were compounded by the fact that the original developers and users who had designed and developed these systems were retiring and the personnel that replaced them did not understand these existing systems. Furthermore, these systems were developed as standalone systems which did not communicate with other related systems. The selected sample system, customer service orders, did not always communicate with other related telecommunication core systems, such as maintenance dispatch, which resulted in many business mishaps. System integration of these disparate legacy systems was an important business goal (Telecommunications Employee B, 1999).

Another goal, besides system integration, was reworked inner functionality and error checking capability. One common problem of many of these legacy systems was that error checking was not implemented during the original design; hence, many errors in output resulted. Since the output of these legacy systems were imported into other systems, these errors were propagated to several related systems. In conjunction with the need for system integration, these legacy systems relied on their own database systems. Hence, if a client put in a customer service order, the customer service order system would use their own data, such as the client's address, even if this data was obsolete and changed to the current client address in other systems, such as in the customer accounting system (Telecommunications Employee B, 1999). Having these legacy systems rely on the same set of current data is one of the most important reasons for system integration. However, the issue of data integrity among this diverse set of systems is outside the scope of this thesis.

Another business goal was to retarget these legacy systems from a mainframe, employee-driven business system to a customer-driven, Web-based system. The move to customer-driven systems was partially motivated due to cost. Rather than have a client request a type of customer service from an employee who would then submit this request as input to this legacy system, much savings could be obtained if these clients requested this service themselves via a Web interface to the retargeted legacy customer service order system (Telecommunications Employee C, 1999).

Replacement systems to meet these new business needs were not a feasible option; consequently, reengineering these systems was considered as the best option in order to handle the new requirements.

In 1998, this telecommunications company was introducing UML as a notation to document the design of new systems. UML, based on earlier notations, was much more familiar to the company's developers than many other notations (Telecommunications Employee C, 1999).

1.1.1.2 Academic Case

Research into problems of legacy systems, which has been ongoing for a long time, was reviewed. Legacy systems often play a business critical role in organisations and their failure can result in serious implications on business (Sneed, 1996). Aebi defines legacy systems as "any information system that significantly resists modification and evolution" (Aebi, 1997). Bisbal identifies several problems that legacy systems may cause business organisations such as legacy systems usually operate on obsolete hardware that is slow and expensive to maintain; legacy software maintenance can be expensive because system documentation and an understanding of the details of a system is often lacking, and as a consequence, tracing faults within these systems is often costly and time-consuming; clean legacy system interfaces are often lacking which makes system integration difficult; and legacy systems are often difficult, if not impossible, to extend (Bisbal, 1999). Newcomb identifies another factor in legacy systems as the need to expand its functionality; hence the need for developers to better understand this system in order to adapt the system to meet new functional requirements without any unwanted side-effects (Newcomb, 2001). In order to address some of these issues, it is necessary to re-document the legacy system in order to help developers fully understand the system in order to make necessary changes and to understand the full set of legacy system interfaces in order to properly integrate these systems with other related systems.

Once the functioning of these legacy systems has been established through re-documentation, it is possible to redevelop the legacy system in order to meet new business needs and the target architecture. However, in actuality, the risk of failure in the redevelopment of legacy system is much too great for business organisations to seriously contemplate such an approach. Additionally, because business and technology requirements are constantly changing, by the time the redevelopment of a legacy system occurs, the redeveloped system is often found to be based on obsolete technology and on business requirements that have already changed (Bisbal, 1999). The Gartner group has shown that a redevelopment approach, based on manually re-writing of code, has success rate of approximately seven percent (Newcomb, 2001). In order to avoid this dilemma, any re-documentation or reengineering tools must be as automated as possible in order to avoid the lengthy reengineering time that redevelopment or reengineering processes that depend on significant manual intervention entail (Newcomb, 2001). Although manual intervention will be needed to refine the final re-documentation effort that fully models the restructured legacy system, this intervention should be subsequent to the re-documentation process when a model of the system has been produced by the thesis' tool and much of the re-documentation effort has been completed.

Other approaches such as wrapping and migration were examined. Wrapping surrounds existing data and applications with new interfaces in order to function in a new technology environment. Wrapping allows the re-use of a well-tested legacy system in which business organisations have a large investment in. The selected legacy system used in this thesis study has used wrapping to some extent where the original card input interface was replaced by Visual Basic screen scrapers. Although screen scraping is popular, wrapping has some drawbacks. Wrapping still retains the static functionality of the legacy system; in order for changes to the system to be made, in response to new business needs, or a proper wrapping of the legacy system to occur, the functioning of the system and knowledge of its interfaces is required. Furthermore, wrapping also has additional problems of overloading and high maintenance costs (Bisbal, 1999).

Migration involves moving an existing, operational system to a new platform, retaining the legacy system's original functionality but causing as little disruption to the existing operational environment as possible during switchover times. Because of the vast number of different legacy systems and the problems that these differences pose, a comprehensive migration approach for all legacy systems is not available but only general guidelines for this migration are provided (Bisbal, 1998).

In order to confirm Bisbal's finding, many of the existing methodologies used to model systems in the telecommunications industry were examined; these modelling methodologies were devoted to real-time, highly-reactive systems such as their packet switching systems (Brinksma, 1986; Bergstra, 1984; Morgan, 1987, Mandrioli, 1985). However, the telecommunications industry, like any other, had other core business systems that were not real-time such as those used for customer-relationship, accounting, and billing. Furthermore, these existing real-time telecommunications methodologies were dependent on highly formalised modelling methodologies that had a steep learning curve and that required much user interaction in order to model their system's events, processes, and responses (Permulla, 1997). Hence, these methodologies would not readily be accepted by this company's management who were concerned with the huge investment that implementation of these methodologies would entail.

Other methodologies, such as Renaissance, which defines a systematic method of software evolution and reengineering in terms of a set of linked tasks and activities, were too complex to satisfy the time and cost constraints of this reengineering effort (Renaissance, 1991). Scott Tilley's methodology of dividing the reengineering process into phases, with guidance for each phase, tends to be too high level to be practical in implementation (Tilley, 1996). Ganti proposed general guidelines in transforming a

centralised mainframe legacy system, like this selected sample system, into a distributed environment, as often found in a Web environment. However, these guidelines require the identification and selection of business processes from the legacy system which would then require development of new applications that retained the legacy data and business logic yet were capable of functioning in the new target environment. However, Ganti's method provides only general guidelines rather than any automated tool; furthermore, the managing of the task of switching over to the new target environment is not made clear (Ganti, 1995).

Because a legacy system already meets some of the business and user requirements demanded of the target system, it is important to understand its operations and interactions. If this legacy system understanding is missing, it can lead to incorrect target system requirements and ultimately to a failed migration process (Bisbal, 1999). The assumption with many legacy system migration efforts is that this system understanding is available through documentation, the expertise of users and developers who have experience working with this system, and other artefacts. If no documentation exists and the human expertise which originally designed the system has long gone, a serious problem in obtaining a full system understanding arises.

In order to overcome the many problems posed by migration such as a wide range of legacy system each with different problems associated with them, it was decided to focus on one particular type of system: a non-real-time, batch-oriented system originally written in COBOL. Additionally, it was decided to focus on one aspect of this migration: system understanding of this legacy system. Since human expertise regarding this system was no longer available and no documentation existed, the only remaining documentation artefact was the legacy system source code itself. Hence, any re-documentation efforts, which would fulfil the system understanding requirement, would have to have their sole basis on the legacy system source code itself. Additionally, it was decided to re-document this legacy system using a more accepted modelling methodology, UML. This modelling would be more easily understood by the developers and they, in turn, would then be able to better understand these existing systems in order to modify them for new business requirements (Booch, 1991).

Unfortunately, the type of source system that UML, in order to base its model on, relies on is an object-oriented, event-driven system (Oestereich, 1999). These core systems were procedurally-structured COBOL systems. Hence, there was a need to restructure these COBOL systems into an object-oriented framework in order for these systems to be modelled using UML. Other benefits can be attributed to an object-oriented framework (see Section 2.5 Object-Oriented Programming) in terms of costs savings, software reuse, et al (Booch, 1991; Jacobson, 1992; Cox, 1991).

Many existing re-documentation tools require an object-oriented system as their source system. Although a few of these tools restructured procedural systems or handled COBOL as their source system's programming language while many other tools did not, the former tools were judged to be inadequate for various reasons given later in this thesis (Harris, 1995; Ball, 1996; Mylopoulos, 1994; Bertouli, 2003).

Because a number of legacy systems were written in COBOL, it was necessary to develop a tool that would restructure and then re-document, through a series of UML diagrams, these COBOL legacy systems. Since my research revealed that such a tool, that met all of the selected system's reengineering requirements, was unavailable at the time, it became necessary to develop one's own tool.

1.1 Existing Research and its Deficiencies: Comparison of Existing Reengineering Tools

In this section, various reengineering tools are outlined along with their advantages and disadvantages. At the end of the section, a comparison of the existing reengineering tools indicate that a gap in existing reengineering tools.

Baumann distinguishes between at least two different reverse engineering tools. One type of tool, syntax-based tools, extracts information from a program on the basis of syntactic information by using variants of scanning and parsing technologies such as parse trees, token lists, etc. Examples of these types of tools are flow chart generators and call graph generators. The tools are limited to syntactic understanding – they extract information about the structure of the program but not about internal dependencies. The other type of tool, semantics-based tools, such as program slicers and data-flow analysers, provide insights into control flow and data dependencies (Baumann. 1994).

Commercially available reverse engineering tools provide a set of limited views of the source code under analysis; however, these tools are an improvement over detailed paper designs in that they provide accurate information derived directly from the source code (Harris, 1995).

Harris has developed a framework to recover custom, dynamic documentation to fit a variety of software analysis requirements. This framework recognizes task-modules within the code but there is no ability to restructure this code into an object oriented paradigm (Harris, 1995).

Currently, most CASE tools, programming environments, and reverse engineering tools have their own proprietary repository. A repository's purpose is to provide an information pool on which tools from different vendors can work. Each repository has its own proprietary structure and because of the lack of a widely accepted standard for the information structure, information can not be shared directly from one tool to another. UML was designed to be this standard. The UML modelling elements can provide the standardised information structure. However, UML provides no rules, algorithms, or heuristics for reengineering operations (Trauter, 1997).

The Maintainer's Assistant was designed to develop a formal specification from old system code. Maintainer's Assistant (Bull, 1995) has the capability for program transformations, code reductions, etc using a typeless wide spectrum language for its intermediate program representation. Maintainer's Assistant has evolved into an industrial-strength re-engineering tool, FermaT. The FermaT transformation system (Ward, 2001) allows transformations and code simplification to be carried out automatically. However, Fermat lacks the ability to restructure code and to provide abstract views of the system using a industry-standard format, such as WSL. Kwiatkowski intended to use his COBOL pre-processor, combined with the use of Maintainer's Assistant and his own custom-designed tool, to extract a specification from COBOL system code and express this specification in Z. A COBOL pre-processor prototype was built using Flex and Bison; his untimely death prevented any further development

(Kwiatkowski, 1998). Neither Fermat nor Maintainer's Assistant are able to handle source code systems whose source code is COBOL (Ward, 1989; Ward, 2001).

Klockwork is a methodology introduced by Masurov in order to extract architectural components from a system's source code. The reason for this methodology, and resulting tool, was that existing tools failed to solve the componentalization problem of large systems by failing to provide a high-level view of the existing code within their component framework. Mansurov, from the extracted component structure, turns this structure into UML packages and objects. Once these components are extracted, manual intervention is necessary to place these components into meaningful packages and classes (Mansurov, 2003). While this tool provides a means to extract an existing architectural view of the system, it does not restructure existing code into a better structure nor does it provide a dynamic view of the system through sequence or activity diagrams.

Murphy has provided a number of tools that scan and apply patterns to source code to extract call graphs, event graphs, and file dependency graphs from source code. These tools are highly dependent on the source code programming language; an example, file dependency graphs are extracted for C programs but not for other languages such as FIELD. Murphy recognized that many reengineering tools are highly-source code dependent. Furthermore, her tools do not restructure the original program (Murphy, 1996).

Zhou recognizes various attempts at object oriented migration of procedural legacy systems such as those of De Lucia (De Lucia, 1997) and Etzkorn which rely on persistent data stores and user documentation respectively for object identification. Zhou's object identification relies on grouping related function calls and their variables together as a cluster; those clusters with less function call dependencies form the candidate objects. No exclusion of externally-declared functions is provided. One of the disadvantages of Zhou's method is that it is highly language dependent; because source code is not represented in an intermediate format, the clustering is highly dependent on the type of data types and structures that are used within the original programming language. Furthermore, her method provides only a method to restructure a legacy system into objects and re-document this restructured system with a class diagram. No comprehensive view of the system is provided through sequence and activity diagrams or through component diagrams (Zhou, 2003).

A tool provided by Zhou was a framework to capture the business workflow of an application by focusing on method invocations, object declarations, and database access statements in order to capture all possible candidates for workflow entities. This framework focused on exclusively on a Java Web server application and provided re-documentation in terms of an abstract syntax tree from which workflow entities were derived. These entities were methods that implemented a particular task within the workflow. One disadvantage with this tool is its Java language dependence. Another disadvantage, recognized by Zhou, was that the extracted business entities were highly dependent on the ad-hoc naming conventions of developers and needed expert user intervention to refine the business logic and entities. (Zhou, 2004).

Tonella provides a method to extract UML interaction diagrams from C++ code. However, since the source system is already object-oriented, there is no need to restructure it from a procedural to an object oriented one. Furthermore, it does not provide static views of the system, such as through class diagrams, or architectural views, such as through component diagrams. (Tonella, 2003).

Zhou proposes a method for restructuring a C legacy system into a C++ system using an intermediate language (IM) that represents, through pattern searching and analyzing, the processes and abstract data types inherent in the source code. Her method uses an object clustering method based on persistent data stores (files) and aggregate data types with the fields of these types forming attributes of their objects. The original procedural form of the code is retained and the code within procedures is represented as its original source code (Zhou, 2003).

Software Refinery parses a number of programming language codes and constructs abstract syntax trees that can be searched using user-defined search patterns. RIGI was intended to provide an abstraction of software representations. This information is provided in a graphical manner that can be summarized, queried, and evaluated. Its main components consist of a parsing subsystem and an interactive graph editor. RIGI analyses and visualises data types, data dependencies, and call dependencies. However, deducing logical subsystems and their functionalities in many systems are difficult due to their complexity and size. There is a need to cluster these dependencies into logical clusters (Mylopoulos, 1994). RIGI was designed to extract, navigate, analyze, and document only the static structure of large software systems, in order to help software maintenance and reengineering activities. RIGI parses, in a fully automatic way, several imperative languages in order to extract artifacts and represents these artifacts as a flat resource-flow graph. The next phase is semi-automatic and allows the user to use pattern-recognition skills and subsystem-composition techniques to represent multiple, layered hierarchies of higher-level abstractions in order to manage the graphs visual complexity (Storey, 1995). However, this phase requires the use to develop their own scripts, based on their knowledge of the system, for artefact extraction. RIGI does not model intra-modular procedural calls nor does it model calls to external system procedures. Furthermore, the propriety abstractions produced by their redocumentation phase were intially unrecognizable by the system developers. RIGI does not rely on industry standard modelling notations for its system visualisation (Wong, 1995).

Moose reengineering environment serves as a foundation for other reengineering tools. It provides a language independent representation and manipulation of source code written in C++, Java, COBOL, and Smalltalk. Moose describes how source code elements such as attributes, methods, classes and namespaces. Moose incorporates both a static analysis of the source code and dynamic run-time traces in its representation of the system. Moose generates its model through abstract syntax trees obtained through parsing; this model is exported to be represented using various external representation formats. Moose assumes its source system is already object-oriented; it does not restructure a legacy procedural system into an object-oriented one. (Bertouli, 2003).

Moore is a tool to help restructure programs from procedural COBOL to object-oriented programs. It requires extensive expert user validation; it requires an expert to approve every step of the restructuring process (Fergan, 1994).

The Refronte tool, by Jarzabek, extracts procedural call and control flow graphs, as well as other software artifacts, from the system source code and stores this information into a database for future use. This tool is able to manage source code from various programming languages, such as COBOL or C++. From COBOL systems, it produces a model of logical record and files to their physical entities in a similar way to a UML component diagram. Refronte also determines classes from records and other data structures. Refronte also restructures C to C++ programs through class identification. However, no comprehensive view of the restructured system is provided through class (static), sequence (dynamic), activity (functional), or component diagrams or

other methods (Jarzabek, 1995). The information, from extracting artifacts from source code, is available in the database but this information is not provided in a comprehensive re-documentation effort.

Egyed focused on dynamic system modeling by adopting a scenario tracing approach where the execution of individual lines of code was traced when a system was executed according to a specified scenario. One problem that was identified with this approach was that different lines of code could be executed for the same scenario depending on the state of the system which resulted in ambiguous traces. Another problem was the great potential for insufficient input data being unavailable resulting in incomplete scenario coverage. Furthermore, this scenario tracing approach usually involved significant manual input and management. Although Egyed's method was focused on C++ source code, this method produced a mixed functional and object-oriented decomposition which resulted in a mixed model of object-oriented, in the form of class diagrams, and functional, in the form of dataflow diagrams, styles (Egyed, 2003).

Van den Brand has developed a reverse engineering tool that is based on the formal semantics of COBOL, expressed in the Mico language, for COBOL74 programs. Although this tool uses an formal intermediate representation of the source code, it is limited to COBOL74 programs and consists of 170 complex rules of the grammar, which leaves out many grammatical rules that may be encountered in a COBOL74 program (van den Brand, 1997c).

Jacobson and Lindstrom discussed a method for manually reengineering an old system to an object-oriented one (Jacobson, 1991). However, no tool to implement this method is provided.

Liu and Wilde developed a series of methodologies for identifying and extracting the object-like features of a program written in a non-object-oriented language (Liu, 1990). No tool to implement this set of methodologies is provided.

Sneed has provided a precompiler along with techniques and tools for augmenting a COBOL program with object-oriented structures. These "objects" simulate object orientation but do not conform to the standards proposed for object-oriented COBOL. Furthermore, these "objects" have a greatly worsened performance compared to their original programs (Sneed, 1988).

One tool, Describe from Embercardo Technologies, has the ability to generate code to represent procedure calls from its tool's generated sequence diagram (Embercardo, 2004). However, these tools require that developers manually code the remaining procedural or class code. Univan and George (Univan, 2000) outline methods to develop a C++ translator for the RAISE specification language. RAISE, although a wide spectrum language, is a typed language, unlike the WSL used in this thesis. Typing of variables is most useful in the specifications for forward engineering, before a particular implementation language has been chosen, and not in reverse engineering when the particular data type of a variable needs to be deciphered from its source code and usage. An example, a Boolean variable can be represented, in some systems, as an integer rather than a Boolean type.

Existing research has produced a number of reengineering tools and methods. A number of reengineering tools/methods, such as Describe, Tonnella, or Moose, produce a limited view of the system, the dynamic and static views, through UML diagrams but do not perform restructuring; instead, they assume that the original source system is already object oriented. Reengineering

systems that restructure procedural into object-oriented systems suffer a number of drawbacks. Firstly, if they rely on source code for their object clustering, their clustering methods are highly programming language dependent, such as relying on the language's particular record structuring and data typing methods; as a solution, Zhou proposed the use of an intermediate language representation for reengineering purposes in order to avoid this source programming language dependence (Zhou, 2003). Secondly, some tools, such as Sneed's restructuring tool, provides non-standard objects while others, such as van der Brand, provides only partial conversion rules for a particular dialect of COBOL. Some reengineering tools are confined to restructuring programs of a particular programming language. An example, the Refronte tool restructures C source programs to C++ only. RAISE, although it relies on the intermediate language representation of a wide spectrum language, depends on typed variables that may be useful in forward engineering but, because variable types may represent different purposes, this typing is not very meaningful during reverse engineering. Other methods, such as those of Jacob and Lindstrom or Liu and Wilde, provide a methodology but no tool. Another aspect of these reengineering tools is that they re-document the system in various ways. Some of these tools provide some degree of restructuring, such as RIGI and Refronte, but produce documentation in a non-standardised format, such as procedural call graphs or program visualisation, which may be proprietary to that particular tool. In the case of RIGI, the redocumentation requires expert user guidance in restructuring. In addition, some tools, such as Maintainer's Assistant, provide no object-oriented restructuring. Egyed does a dynamic trace of the system execution in order to produce documentation yet he realises the shortcomings of this method in terms of insufficient data available for all scenarios and ambiguous traces. Furthermore, Egyed's documentation of the reengineered system is a mix of object-oriented and functional style diagrams. If a systematic form of documentation, such as UML, is used and extracted from the system, a limited view of the system is provided by existing tools. Describe, Moose, and Tonella extract class and sequence diagrams from object-oriented systems to provide the static and dynamic views but they do not provide diagrams, such as those of activity, component, or deployment diagrams, to represent the functional or architectural views of the system. No existing tools restructure a procedural to an object-oriented system, via an intermediate language representation, and then re-document this restructured system with a full system view (as defined as encompassing the structural, behavioural, dynamic, and architectural views of the system) using a systemised method of documentation.

1.2 Problems to be Researched

In order to represent a batch-oriented COBOL system at a higher level of abstraction via UML diagrams, a study must be performed on the selected sample legacy system, whose characteristics are that it is a batch-oriented COBOL mainframe-based system whose only artefact is the source code and that no information is available to determine the problem domain that the system models either through human actors or documentation, what UML diagrams can be extracted using what methods and under what conditions. Another research area within this thesis, in order to facilitate a WSL-UML notation which would enable WSL to represent high-level modelling abstractions, is to define a set of WSL constructs that correspond to UML diagrammatic notation.

Because the system under investigation is a batch-oriented system, with a very limited or non-existent set of external events, certain diagrams, such as statecharts, were deemed to be infeasible to extract.

Although the source code is parsed and a subset of UML diagrams is extracted from them, the extraction of semantic knowledge within the source code is outside the scope of this thesis. An example, developer comments are not extracted from the source code and then within the source code for any knowledge of the purpose or design of the program.

The object classes that are obtained through restructuring, namely through the defined object identification algorithm, correspond to a one class to one object relationship. This one correspondence does not correspond to a true object-oriented system with class hierarchies, class inheritance, or polymorphism. Justification for this approach is provided in the Class Diagram chapter.

The chosen intermediate language, WSL (Wide Spectrum Language), is, by its nature, typeless. This typelessness presents a problem when trying to accurately represent the legacy system from its COBOL programming language to its intermediate representation in WSL.

1.3 Research Questions

After this feasibility study has been conducted and the results analysed, the following research questions can be answered:

1) Given a procedurally-structured, batch-oriented COBOL system, is it possible to develop algorithms, whose output is validated through a UML checklist to ensure diagram syntactical correctness and representational accuracy, to re-document this system, by extracting from the source code, a series of UML diagrams that represent the structural, behavioural, dynamic, and architectural views of the system?

2) Does this re-documentation effort entail restructuring and entity identification/determination processes? If so, is it possible to develop algorithms, validated through various means, that handle these processes and how can the information, obtained during these algorithms' execution, aid in the extraction of these UML diagrams?

1) If this system could be re-documented, is it possible to demonstrate, through parallel case studies of other re-documentation/reengineering tools, that this re-documentation of the system, by achieving a better system understanding, increases the effectiveness of the developers in maintaining this system?

1.4 Research Methodology

While pursuing the research outlined in this thesis, the following research methodology was followed:

1) Problem: A problem was observed; this problem was a legacy system (in particular, batch-oriented legacy systems whose only remaining information artefact was the software code of the system itself) which was in need of re-documentation and restructuring. Incremental maintenance changes over the years had obfuscated its original design. A business case had been made to find ways to re-document these batch-oriented legacy systems from their source code in order for developers to understand their present structure and behavior both in order to modify these systems to meet integrate them with existing systems or move them to new platforms. By finding ways to re-document these types of systems under these constraints (existing source code is the only artefact) and determining the methods and the conditions under which such re-documentation might occur, a proposed solution to this lack of documentation was discovered.

2) Hypothesis or Proposed Solution: A hypothesis was developed that re-documentation could be obtained, with constraints, from the batch-oriented system's source code via a selected sub-set of UML diagrams. Re-documentation was necessary in order to provide up-to-date system documentation that could be used by the developers for future redevelopment of the system. UML diagrams were selected for several reasons (outlined in the thesis). The novelty of the thesis' approach is that while UML was becoming the de-facto standard for system documentation, UML tended not to be used to re-document batch-oriented legacy systems.

3) Methodology: In order to re-document such a system through a selected series of UML diagrams, it was necessary to restructure the existing system. UML is purely based on an object-oriented model; the original legacy system was procedurally structured and driven. This methodology consisted of developing algorithms and the consequent tool to automate the restructuring and re-documentation of a legacy system. This methodology consists of two parts:

   i) Literature surveys of secondary, related research to determine what work has been previously been done with its successes/failures, algorithms that were used, tools already in existence with their potential for adoption were conducted in order to determine what work had already been done in the restructuring and re-documentation research areas. In several areas, such as extracting UML diagrams from source code, very little work had been done; in comparison, a body of literature existed for restructuring procedurally structured code.

   ii) With this knowledge of past work, experiments were designed in order to determine if these algorithms already existed (such as for object identification and if these algorithms were best suited, given a set of criteria, for the selected system's restructuring efforts). Often, these existing algorithms had to be modified in order to best suit the selected system's restructuring needs, given this set of criteria (Millham, 2002; Millham, 2003a). In other instances, where the body of literature in this area was rather sparse such as UML diagram extraction from source code, algorithms had to be devised in order to accomplish this goal. When algorithms were designed, a set of criteria for their success were devised and experiments conducted in order to determine whether these criteria were met. This experimental stage consists of two phases:

      1. Small case phase: criteria for success and test cases for experiments are designed and algorithms are developed to handle small sample of code (small cases). The selected test cases are run and the results are verified against the pre-defined criteria for success. Often, the experimental results indicated that these criteria were not met and the algorithm had to be revised and new experiments conducted using the revised algorithm until the success criteria were met.

      2. Large case phase: once the algorithms have been verified to be correct against the pre-defined success criteria and after these algorithms have been incorporated into the tool, the algorithms, within the tool, are executed against a larger example, representative samples of the legacy system, and the results produced from this run are verified using various methods as outlined in this thesis.

4) Results: the final step in any research methodology is validation of the experimental results against the pre-defined success criteria in order to determine the success or failure of the experiments. In many ways, validation is similar to determining whether the experiments met a selected set of criteria. An example, the object identification experiments in step 3 determined which algorithm, including the specified revised algorithm, met the selected criteria the best. The criterion for the object identification experiment was which algorithm produced a set of objects with the most intra-coupling between members of an object and had the least inter-object coupling.

Other methods of validation for the selected object identification and consequent restructuring of the selected legacy system was used. The event identification experiment (see Event Identification Experiment in Appendix E) confirmed that most method invocation events occurred and were handled within their own class with few inter-class method invocation events. This experimental data confirmed that the object identification algorithm was correct in that it placed highly-coupled procedures together in the same class and, thus, reduced inter-class dependencies. A hand-proof was supplied (see Section 6.6 Correctness Validation of Object-Oriented Restructuring) in order to prove that the functionality of a system, in accessing and manipulating data via methods and data members, remained unchanged provided that the renaming of these methods and data members, from global variables to a class structure was done in a consistent manner.

The independent task evaluation experiment was validated as to its correctness using a small but representative sample input program whose results could be determined with exactness if these inputs were executed in their original strictly sequential manner. The algorithm evaluated this sample program as to their task independence. Tasks were defined at both the individual code line and procedural granular level, in conjunction with their later use in the extraction of UML diagrams. The defined algorithms evaluated these tasks, at both of these granular levels, and evaluated which of these tasks could be executed in parallel and which, due to data or control dependencies, had to be executed sequentially. Tasks that were deemed to be able to execute in parallel where executed in a pseudo-parallel manner by having their original sequential order reversed; the logic is that if two tasks can execute in parallel then it would not matter if the task that was originally assigned to execute after the first task was executed before the first task. If the orders of tasks that can execute independently are ordered in an independent manner, without regard to their original sequential execution order, and the outputs of this program are functionally equivalent to the original sequential executing program, the independent task evaluation algorithms can be verified to be correct. Intermediate calculation results, during the independent task program's execution, are also evaluated and compared with the results obtained during the sequential executing program in order to ensure that, by some chance, a set of independently executing tasks do not produce the same final results as the same program executing in a strictly sequential manner.

After the restructuring steps have been validated, the generated UML diagrams, whether in WSL-UML or in UML graphical notation, are validated in the several ways. The WSL notation is developed in strict accordance with OMG's UML specification 1.5. Each construct with its members has a reference to the UML specification document for that diagram. Additional fields, such as in the activity diagram, are used to aid viewer's comprehension but do not detract from the correctness of the defined WSL UML constructs. By adhering to this strict correspondence, the defined WSL constructs can be ensured to be truly representative and semantically equivalent to the UML diagrammatic notation. Secondly, a checklist is used to ensure that the generated UML diagrams are well-formed and acurrately represent the restructured WSL representation of the selected COBOL system. Thirdly, code generation (see Appendix D) can be used to generate a restructured system using the generated WSL UML constructs. An example, the extracted class diagram from the selected source code is used to generate a class structure during the, albeit limited, generation of C++ code.

1.4.1 Overview of Research Process

After a survey of existing tools demonstrated that no tool was available that could meet the above business problem, it was decided to develop a tool that could address this business problem. In order to address this problem, a methodology or plan of action had to be instituted and followed.

A literature survey was conducted in order to reveal current research in reengineering legacy systems, particularly COBOL legacy systems. Different methods of re-documentation were examined.

Because UML had been already adopted by this telecommunications company and was becoming in widespread use, it was decided to adopt this modelling methodology over many others. UML, by its nature, is an abstract, imprecise modelling notation. Yet because it is built on earlier modelling notations such as state transition diagrams, it is much more familiar to developers than other notations (Rumbaugh, 1991). Furthermore, the abstract, imprecise nature of UML is also an advantage. Although business processes should be modelled precisely in many industries, such as the airplane flight navigation software system industry, where imprecision is both expensive and deadly, in many industries the business requirements often are too fluid, changing, and imprecise to be modelled feasibly using a precise modelling notation given the lengthy time requirements of precise modelling versus the required deadlines of its software development. By providing an abstract and imprecise view, changing business processes can be remodelled using UML without as much effort as if these processes were modelled using a more precise modelling notation.

However, in order to adopt UML as a modelling methodology in reengineering this system, the present procedurally-structured batch system had to be restructured into an object-oriented, event-driven system (OMG, 2004). Furthermore, parallel and sequential flows of control and data had to be determined from this source code in order to model these flows as such in various UML diagrams, such as activity diagrams. Possible events had to be identified from the batch-oriented system in order to represent them in UML diagrams such as sequence diagrams (OMG, 2004).

In order not to limit this tool to just COBOL, it was decided to convert this tool into an intermediate representation, WSL. WSL was adopted, as an intermediate language, because of its tool support, its platform independence, and a growing body of literature on software evolution using this language (Ward, 2001).

WSL, because it is a wide spectrum language, can express the abstract, if imprecise, constructs of the UML diagramming components yet is also able to express the detailed level needed to represent programming language constructs. One of the chief arguments against WSL is that it is type less (Ward, 1992). Although this typelessness gives WSL platform independence, it also presents constraints when trying to express variables in terms of their data types.

However, data typing of variables or constructs is not as straightforward as it may seem. An example, COBOL data types may be very precise (giving both the base and mantissa maximum sizes of a number) yet they also are very general. An example, a record consisting of number and string fields may be transferred into a purely alphanumeric record. There is no strong data type enforcement in COBOL (Feingold, 1981). This type of data typing, along with the lack of strong data type enforcement,

presented problems when trying to convert COBOL into WSL and then from WSL to C++. The original data types from the COBOL program were retained in the database for conversion to another programming language data type.

Furthermore, it is difficult to translate directly the data typing used by COBOL into the primitive data types used by UML. One example might be the way that a COBOL record of purely alphanumeric characters can hold numerous fields of varying data types (Feingold, 1981).

Also, many languages lack the type of typing used in UML notation. An example, UML has types of class, sequence, etc which many languages lack. In order to ensure that this intermediate language has all the typing inherent in a UML model, one would have to develop a much more specialised language whose data typing was a subset, at the least, of the UML entities (Ideogramic, 2004).

In order to convert COBOL to WSL, a number of approaches were tried. COBOL has a huge number of possible constructs and a large number of dialects (van den Brand, 1997a). Another factor hampering the development of translation tools is that COBOL grammars are proprietary; few freely-available COBOL grammars, which can be used as the basis for translators, are available. One freely-available COBOL grammar, which was obtained through reverse engineering of COBOL systems at the Free University of Amsterdam, was in a different COBOL dialect than that written in the selected sample COBOL system (van den Brand, 1997b). Similarly, previous attempts to translate COBOL into WSL using YACC by Jan Kwiatkowski produced a half-developed tool. However, this tool assumed a different COBOL dialect as the basis of its development than that of the selected COBOL system. Rather than spend a large amount of time reworking these existing tools in order to handle the existing COBOL dialect of the selected sample system, it was decided to manually convert this sample system from COBOL to WSL. Given the huge number of COBOL constructs (many of which perform the same function such as the constructs of EQUAL or GIVING), the many different COBOL dialects, and the lack of freely-available COBOL grammars that could be used by compiler-compiler tools to generate a COBOL to WSL converter, it was felt that automating this task would involve too much extensive work.

This restructuring required a literature search to determine what work had been done to identify objects and events from legacy systems, particularly COBOL systems. Different algorithms were tried to identify the algorithm best suited to identify objects with the least amount of inter-class coupling (Millham, 2002). Similarly, a literature search was undertaken to determine what work had been done and algorithms used to identify possible independently-executing tasks. Different algorithms, using different levels of task granularity, were used in order to determine algorithms that best produced the greatest degree of task independence (Millham, 2003a). The information gained during these experiments was used to develop algorithms that extracted information from the source code, such as objects and the sequence of their interactions. This information was then incorporated in the extraction of UML diagrams.

After restructuring of the WSL representation of the legacy system was complete, a literature search was made in order to find out related work in automatically extracting UML diagrams from source code. Unfortunately, unlike restructuring legacy systems, this literature search produced few results. Algorithms were developed, based on information gained during the system's restructuring process and from knowledge of UML, to automatically extract these UML diagrams.

UML diagrams, along with their components, were then expressed in their WSL equivalents. By expressing UML in WSL notation, there is a direct mapping between the platform-independent, mathematical, intermediate language of WSL, along with the legacy system that it is represented in, and the abstract modelling notation of UML. From their WSL notations, this information was exported, via XML, in order to be imported and converted into a visual UML diagram representation using the Poseidon diagramming tool (Poseidon, 2004).

1.5 Original Contribution

The main focus of this research is to develop and experiment with a method, upon which the thesis tool is based, to represent a batch-oriented COBOL system at a higher abstraction level using a series of UML diagrams. In order to accomplish this, a feasibility study will be undertaken with a selected sample legacy system, a mainframe batch-oriented COBOL telecommunications system, in order to determine which Unified Modelling Language diagrams can be extracted satsifactorily from this system and the conditions under which this extraction of diagrams can occur using source code as the only artefact.

In order to extract these diagrams and represent them in a WSL-UML notation, WSL must be extended in order to represent abstract UML modelling components such as class, association, and activities. These WSL extensions, whether in the form of linked structures or lists, are directly mapped to their UML 1.5 specification equivalents in order to provide direct UML-equivalent mapping of the transformed re-documented system through the tool's generated diagrams to the WSL-UML and UML graphical notations.

The selected sample system was developed as a standalone batch-oriented mainframe system in an ad-hoc manner with no documentation and a long history of incremental maintenance changes. Due to mergers and the loss of personnel through layoffs, retirements, and deaths, the original developers and user community that specified and developed this legacy system are gone. The only remaining artefact is the source code. Relying on this source code, this system must be first restructured and then re-documented, at a higher level of abstraction, via a series of UML diagrams.

The methods developed in this thesis and the thesis tool, TAGDUR, which incorporates these methods, is unique for the following combined reasons:

a. it provides structural (class), behavioural (activity), dynamic (sequence), and architectural (component and deployment) views through its generated UML diagrams

b. many existing comparable tools provide only structural (class) with a few tools providing a dynamic view

c. restructuring of a procedurally-structured system to an object-oriented one. Many existing tools restructure procedural systems to an object oriented systems but few of these tools also re-document the restructured system. This redocumentation is not in a systemised, industry standard notation but rather proprietary to the particular tool.

d. many comparable tools do not perform restructuring of this system but rely, for their source system, on an object-oriented source system

e. although the restructuring/entity identification/determination algorithms are not unique (global analysis with functional pattern identification [procedures with high fan-in and fan-out]), these algorithms provide information that is directly used in the extraction of UML diagrams. Restructuring of the source system also aids maintenance and re-deployment efforts

f. the UML diagram extraction algorithms have a use in providing fine-grained system level detail that can be used to express both business rules and program design that help aid maintenance (business rule review and better system understanding)

g. intermediate representation of the source system is in WSL with the following advantages:

    i. formal notation

    ii. can express low-level program code (assembly language) but is extended in thesis to represent higher-level modelling notations such as UML

    iii. proven set of program transformations available for WSL through Fermat and Maintainer's Assistant

This thesis tool addresses a gap in existing reengineering tools and/or methods in that, at present, no existing tools restructure a procedural to an object-oriented system, via an intermediate language representation, and then re-document this restructured system with a full system view (as defined as encompassing the structural, behavioural, dynamic, and architectural views of the system) using a systemised method of documentation, such as UML. The methods outlined in this thesis and implemented in this thesis address this gap.

One of the goals of this restructuring and re-documentation of this legacy system is enterprise integration and redeployment on a new platform. The original system was not designed to interact with related systems, such as those of troubleshooting or accounting. In order to integrate this system, the developers must fully understand this legacy system's structure, behaviour, and interaction with its components in order to integrate it with other related systems. Additionally, the original system was designed to be operated by telecommunications employees, in a batch mode, on a mainframe system. Changing business needs required that interaction with this system be customer based via a Web-based platform. Moving from a monolithic mainframe environment to a Web platform usually requires a restructuring of the system from procedural to object-oriented and event driven (Harris, 1995). It is this type of restructuring that occurs with this sample legacy system. In Chapter 11, some specific examples are provided as to how this redocumentation of the system can help with this migration to a Web environment.

A set of rules was formulated to transform a batch-oriented system, represented in WSL, from a procedural to an object-oriented, event-driven system. These transformation rules have been validated to be correct through various means. This thesis' tool, TAGDUR, is rare in that few tools have the ability to transform a procedural to an object-oriented, event-driven system. Furthermore, this tool is unique in that it has the ability to identify independent tasks from source code at the individual code line and at the procedural level of granularity.

A set of rules was formulated to translate a small subset of COBOL constructs into WSL constructs. These rules provide the ability to translate a programming language program representation, in this case COBOL, to an intermediate language. Several advantages of using an intermediate language for re-documentation and transformation purposes, as opposed to using the original programming language, are outlined further on in the thesis.

Automatic generation of documentation, in the form of UML diagrams, of this restructured batch-oriented system is produced by the thesis tool, TAGDUR. These UML diagrams represent both the static, behavioural, architectural, and dynamic views of the system. These UML diagrams provide re-documentation of the system in order to enable developers to better understand the existing system, which has no existing documentation, in order to help the redevelopment or redeployment and to provide an aid for business analysts to understand and review the business rules embodied within the source code (Aiken, 2000).

1.6 Structure of Thesis

Chapter 1 gives the background of the legacy system being reengineered, scope, and original contribution of the thesis.

Chapter 2 provides an overview of software engineering, program visualisation and program understanding, and formal methods.

Chapter 3 provides a background on two of the major languages used within this thesis: WSL, a formal mathematical intermediate language, and UML, a modelling language.

Chapter 4 outlines the thesis' proposed approach along with its restructuring and UML diagram extraction algorithms. Extensions to WSL are outlined that enable WSL to abstractly express syntatically equivalent UML components.

Chapter 5 outlines a set of rules to translate COBOL into the Wide Spectrum Language (WSL).

Chapter 6 investigates a clustering technique to group data and the operations that manipulate them into classes and objects.

Chapter 7 investigates several methods, based on different granular units, to determine task independence. These methods are then used to identify independent and sequential tasks among the class operations. This chapter also outlines a method to identify pseudo-events from procedural code and then outlines a method, based on the determination of task independence method, to determine the asynchronicity or synchronicity of these events. In addition to evaluating tasks for independence on a procedural granular level, this chapter outlines the algorithms used to extract a sequence diagram from the source code and the results of this procedural granular-level independent task evaluation algorithm.

Chapter 8 outlines a set of rules and algorithms o derive UML activity diagrams from this transformed source code.

Chapter 9 outlines a set of rules and algorithms to extract UML deployment diagrams from this source code after transformation.

Chapter 10 lists several UML diagrams, such as collaboration, use case, and statecharts, and explains why these UML diagrams can not be obtained from the source code of a batch-oriented system.

Chapter 11 gives a brief overview of the tool, TAGDUR, and its design and functionality.

Chapter12 discusses the results of this investigation and whether this investigation met the required criteria for success. Possible future work, uncovered during this investigation, is mentioned as well as the conclusions of this thesis.

Appendix A lists aCOBOL source sample of selected legacy system.

Appendix B lists the WSL code translated from the COBOL source sample.

Appendix C models the selected sample of WSL code, after program transformation, using UML diagrams.

Appendix D lists the conversion rules, plus a sample, for WSL to C++.

Appendix E provides the experimental data derived from event identification using the selected WSL sample(s).

Appendix F lists, by diagram type, the corresponding WSL-UML notation of UML diagrams.

Appendix G provides comments from the local supervisor after evaluating the thesis results, notably on the correspondence of the original COBOL samples to its WSL representation and, based on his experience, the financial benefits of this reengineering to industry.

Appendix H validates the UML diagrams, generated by the thesis tool, using a UML checklist

Appendix I lists publications by the candidate.

Chapter 2: Background

2.1 Introduction

In this chapter, a brief history of software development is given that details the growing need to address the complexity and design obfuscation created through incremental maintenance changes of legacy systems. The topics of reverse engineering and object orientation are introduced as techniques to reduce the complexity and increase the program understanding of legacy systems.

2.2 History of Software Development

The history of the software development industry was traced by Bergland through the decades, outlining the environments in which software developed and the problems produced by each type of software environment. During the 1950's, software development was a "cottage industry" which produced programs containing a large amount of undocumented code which was maintained by the people who originally wrote it. Although the coding result of this "cottage industry" was inflexible and inextensible code, the coding result was quite adequate for the demands of the time (Bergland, 1981).

During the 1960's, a crisis arose in software development because software became two orders of magnitude more difficult and software personnel were constantly changing so that the original software developers were no longer the ones maintaining it. The structured programming methodology of the 1970's was primarily an attempt to address the problems of large-scale programming. The methodology began in response to rapidly rising costs of software and to the feeling that a change to the software development process was technically feasible. An example of the rapidly rising cost of software was that more than 1% of the Gross National Product of the U.S.A. was beginning to be spent on software in the U.S. (Bergland, 1991).

Booch's research indicated that functional development methods suffer several fundamental limitations. These methods:

- Do not effectively address data abstraction and information hiding.
- Are generally inadequate for problem domains with natural concurrency.
- Are often not responsive to changes in the problem space (Booch, 1991).

2.3 Reverse Engineering and Reengineering

One of the reasons for software renovation of legacy programs is that these programs contain an enormous investment and knowledge of the application domain; consequently, it is economically worthwhile to renovate the system containing the application solutions and to use the resulting logical and data structures (Fergen, 1994).

Existing programs require constant adaptation and functional enhancements in order to meet their changing internal and external demands. The activities involved with these enhancements are called software maintenance. Software maintenance is a difficult task because old programs are insufficiently structured and very often no reliable documentation exists for them. Software engineers must be able to rebuild the overall design of a piece of software as close to possible to the original design of the software while taking into account incremental maintenance changes (Baumann, 1994).

Important business rules are often embedded deep within the program code. Aiken has explored the issue of extracting business rules from legacy source code as part of a system migration and transformation process. These business rules can be used as part of the round trip engineering effort where the rules are incorporated into the target system (Aiken, 2000). During the evolution of software, changes are applied to the source code in order add function, fix defects, or enhance quality. In systems with poor documentation, source code becomes the only reliable artefact of the system. Hence, the reverse engineering process has tended to focus on code understanding (Muller, 2000a; Aiken, 2000).

Reverse engineering is touted as the preferred way to overcome the obstacles posed by maintenance. Various structural information, which was obtained during reverse engineering process, enables the developer to obtain different views of the program. Using graphical and textual notation, implied dependencies and relations of the program are made visible and enable the maintainer to obtain a stepwise refined understanding of the program (Bauman, 1993: p 94).

Chikofsky defines reverse engineering as the process of analysing a system in order to identify the system's components and their inter-relationships and to create representations of the system at higher levels of abstraction (Chikofsky, 1990). Reverse engineering involves taking the mass of concrete implementation details from the computer representation and creating an abstract representation from them (Ashrafuzzaman, 1995). In the outlined research, a mass of source code from a legacy COBOL system is taken and an abstract representation, in the form of a UML model, is extracted from them.

Typically, reverse engineering analyses the source code of a system and builds models of it on higher level of abstractions. These models may be of the same type as used in forward engineering. Forward engineering is the stepwise refinement progressing from analysis to design and then to implementation, resulting in source code. Reverse engineering is the stepwise refinement progressing from the implementation (or the source code) to the design model and then to the analysis model (Trauter, 1997).

Mathematically precise functional specifications may be a useful concept but unfortunately, in many parts of the IT industry, it is seldom used. Corporate managers want their business analysts to focus on their core strengths, defining business processes, rather than developing excruciatingly detailed and mathematically precise functional specifications (Owen, 2004). Aiken has explored the issue of extracting business rules from legacy source code as part of a system migration and transformation process. These business rules can be used as part of the round trip engineering effort where the rules are incorporated into the target system (Aiken, 2000).

One problem that Own identifies in refactoring existing systems is that the existing rules embodied in such a system are rarely, if ever, entirely correct. These inconsistencies may create problems in themselves as analysts work to resolve difficult business logic issues that they otherwise could avoid dealing with (Owen, 2004). By visualising the business rules as part of the control constructs within the source code in various UML diagrams, it is hoped that business managers, and associated personnel, would be more easily able to review the business rules already encompassed with the source code and modify them if necessary (Babcock, 2005).

As software systems undergo maintenance changes, each change potentially moves the system away from its original design. As maintenance increases, the system becomes increasingly more difficult to understand and to modify properly. In order for these heavily-modified systems to survive, they must be repaired or reengineered. In order to help make maintenance easier, it is necessary to understand the system's components and how they interact (the architectural view of the system). However, there are several different views of a system's architecture. For example, the concrete architectural view of the system depicts how the components in the code interact while the conceptual view of the system's architecture portrays how the components are structured. Many tools, such as Acacia, Rigi, and PBS extract information from the source code of how the low-level components of the system interact. However, Holt argues that in order for a developer to understand the system, he must also understand the system hierarchy of a system. This hierarchy is defined as how the modules of a system are grouped into subsystem and how these subsystems are grouped into higher level subsystems. Information regarding this hierarchy can be obtained by interviews with those familiar with the software, file naming conventions, program structure information, directory information. Component interactions can be determined through analysing procedure calls (Holt, 2000).

2.4 COBOL-Based Software Systems

Some of the difficulties encountered while trying to reverse engineer a system are that the programs to be reverse engineered often have poorly-structured source code, missing or incomplete documentation, and a large number of ill co-ordinated maintenance changes over time. Some additional reverse engineering problems specific to COBOL legacy systems are the large number of possible COBOL constructs and the many COBOL dialects which make it infeasible to create a "universal" COBOL converter tool (van den Brand, 1997a). Another problem is that COBOL programs are, to some degree, machine dependent and dependent on particular peripheral devices; these dependencies make the system's reengineering to a different platform more difficult. An example, moving a COBOL program from one machine to another involves not only re-compilation but also the rewriting of at least one section of the COBOL program (its Environment division which describes its particular target machine) (Feingold, 1981).

Van Deursen undertook a study of COBOL legacy systems. He discovered that more than half of the code contained data stucture declarations which defined how record structures are mapped onto memory. The remaining code contained procedural code, which contained the business logic of the program. Because of the limited abstraction capabilities of COBOL, this business logic is difficult to isolate from the rest of the code. Within a COBOL program, all variables are global but programs are divided up into paragraphs, or procedures, with no parameter-passing mechanism (van Deursen, 2001).

A typical COBOL program is 1500 lines of code. Programs may call one another using the COBOL **CALL** statement and utilise the COBOL copybook mechanism to include other program files within it. COBOL legacy systems are typically of 500 000 lines

of code, sub-divided up into 200-500 programs. Batch processing is typically initiated by batch jobs, most commonly written in JCL (van Deursen, 2001).

Van Deursen compared COBOL-based legacy systems with those of UNIX-based C systems and determined two important differences. The first difference was that COBOL systems were much more data-centred. The second difference is that COBOL is much more limited, having no parameter-passing mechanism procedure or function calling mechanism. While COBOL has no pointer structures, which simplifies data flow analysis, it has the REDEFINE construct where multiple record layouts can be assigned to a single memory area, which complicates single record definitions. In practice, extraction techniques that were developed with C legacy programs in mind are difficult to reuse in a COBOL setting (van Deursen, 2001).

Van Deursen has identified the need for having a documented architecture of a legacy system when companies merge, when new developers are added to a maintenance team, or when legacy functionality needs to made available in new paradigms, such as object oriented systems. In order to represent this extracted architectural structures, graph visualisation, zooming in and out of abstractions, and the ability to see the relationships between the extracted structures is essential. Clustering analysis can help replace traditional functional decomposition with an object-oriented decomposition. Mainframe-based legacy systems poses particular extraction requirements, such as the requirement to take a data-centered point of view (van Deursen, 2001). These differences in COBOL program versus other types of programs make a generic reengineering approach to legacy systems, without the use of an intermediate representation, very difficult. In addition, these differences must be taken into account while developing a reengineering approach.

2.5 Object-Oriented Programming

Jacobson defines object-orientation as a technique for system modelling which involves modelling a system as a number of objects that interact (Jacobson, 1991: p 42). Objects are related to the real-world objects that are modelled within software systems. An example, the real-world objects of a Bank Account and Bank Transaction may be incorporated as the objects of BankAccount and BankTransaction within the software system. The common characteristics of these real world objects such as BankAccount having an account balance or performing an operation such as updating this bank balance become the attribute of AccountBalance and the method UpdateAccountBalance of this object respectively (Jacobson, 1991). Because the objects have been built upon more stable intermediate forms, such as the real world domain, these objects, and their corresponding design, are more resilient to change than non-object-oriented methods (Booch, 1991).

Because objects within the software system correspond to their equivalent objects in the real word, the semantic gap between understanding the business processes being modelled in the software system are theoretically reduced. Furthermore, because objects encapsulate related variables and procedures together within their class structure and restrict access to these procedures/variables, modifications to an object-oriented system tend to be localised without unforeseen side-effects outside the object (Jacobson, 1991: p 43).

Object orientation methodology has been derived from earlier methodologies such as structured methodologies and, as such, are more familiar to older developers who are acquainted with these older methodologies (Booch, 1991). By packaging a system into

logical packages or components, a developer can more easily understand a component than if they were required to understand the larger system as a whole (Cox, 1991). Significant improvements in productivity and code quality have been found when developers adopted the object oriented paradigm rather than the earlier structured methodologies. Object oriented software tends to be smaller than their equivalent non-object-oriented implementations due to modular re-use. This smaller size means that there is often less code to develop and maintain but also translates directly into cost and schedule savings (Booch, 1991: p 71; Flint, 1997). Other advantages of object orientation are that the object model encourages software reuse both at the software component and at the design level (Booch, 1991). Furthermore, object oriented development, both at the implementation and testing stages, can occur in steps during the software system development process rather than the traditional "big bang" approach. This incremental development reduces the development risk of complex system implementation (Booch, 1991).

Object orientation within COBOL produces programs that are more compact and better suited as to their intended purpose. An object oriented program can be characterized by a substantial independence of hardware and software suppliers and a greater use of reusable parts. Object orientation is more reliable, easier to maintain, and to adapt to new functionality (Fergen, 1994; Meyer, 1987). There is currently a considerable interest in migrating extant code to more modern paradigms such as object-oriented (Harris, 1995).

Object orientation is applicable to legacy systems where object orientation promises to allow for easier maintenance, reuse, and integration with other applications in a network-centric environment. The move to an object oriented paradigm is particularly important in regards to legacy systems. Legacy systems can be defined as mission critical software systems that contain comprehensive business knowledge and constitute large assets for organisations; however, due to maintenance activities, their quality and operational life are constantly deteriorating. With rapid technological changes, there is pressure to migrate or port these existing systems into modern platforms, such as object-oriented systems (Zhou, 2003).

One of the problems that this real world object to system object mapping represents in reverse engineering is that during reengineering, without clear information about the world domain that the software system models, the identification of software objects, along with their hierarchy, is very difficult, if not impossible to extract from source code alone.

2.6 Restructuring Procedural Legacy System to Object Oriented Systems

Restructuring a procedurally structured and driven legacy system to an object-oriented, event-driven system involves a number of steps. These steps include identifying possible objects from existing procedures and variables within the source code, determining the execution independence of tasks (of various granularities) of the system, and identifying events from the source code. In this section, some research background of each of the restructuring steps is provided.

2.6.1 Background of Program Function Abstraction

In this subsection, a tool, IBM COBOL Structuring Facilty, is outlined as a means to restructure COBOL systems. The automatic abstraction process of this tool is used to represent an abstract view of the system as well as provide an extraction of the business rules. COBOL systems have reengineering tools, some of which utilise program function abstraction for program understanding and to enable software reuse. The Linger/Mills/Witt structure theorem guarantees that any flowchart can be transformed into a structured programming flowchart (Linger, 1979). This theorem has been embodied in a reengineering tool, IBM COBOL Structuring Facility, which automatically restructures unstructured COBOL programs into a structured program format for improved maintainability (Hausler, 1990).

The automatic abstraction process is a multi-step process. First, the unstructured COBOL is restructured into a structured program. Then, the restructured code is run through an automated abstraction facility. This facility analyses the program data in order to localise the scope of each program variable and to split occurrences of overloaded data items. Thirdly, the facility computes the function of each prime during the functional expansion of the structured program. For looping primes, this consists of program slicing to allow the abstraction of the prime one iteration at a time. Furthermore, program slicing allows the recognition of recurring loop pattern within code (Hausler, 1990).

In order to abstract a program function, it is necessary to determine exactly what a program does to data in every possible circumstance and to precisely describe this function using the algebraic structure and mathematical properties embodied in a structured program. High-level abstractions of the program function describe the business rules of the program. An example, certain rules for applying a price discount depending on the customer type may be embodied within the code; these rules must be extracted from the code and expressed in non-procedural terms for business analysis (Hausler, 1990).

Hausler argues that program-function abstraction, which provides mathematically precise definitions of the functions of programs, provides advantages to many groups involved within the software process. An example, program function abstraction, by providing at every level of abstraction the precise functional effect of programs, is able to provide the maintainers of the program with information to evaluate where and how the requested changes should be implemented. Once this program is modified according to the requested changes, the newly-changed program is re-analysed to determine whether the modifications provide the desired effect and show no unplanned side-effects. Developers could use program-function abstraction to ensure that their program, which is under development, meets the specifications. Program-function abstraction is useful for providing functional documentation as well as a way for future programmers to find pieces of code that provide a precise functionality that they require for a future program and by thus doing, program-function abstraction aids software reuse (Hausler, 1990).

Data usage is analysed in order to reformulate this usage according to compact and precise abstractions of all data references and in order to determine data anomalies, such as un-initialised data reads. Program function abstraction for each prime is handled differently depending on the type of the prime – sequence, alteration, or iteration. For sequence, a trace-table technique is used that records the effect on the state space of all variables, with one table column per variable. In the case of conditional rules, a trace table analysis is performed for every possible combination. In the case of alteration abstraction (if-then-else), the trace table analysis eliminates impossible cases. With iteration primes, no single technique produces the best results in all cases but a generalised table-lookup, pattern-matching approach produces the best approach (Hausler, 1990).

2.6.2 Background of Object Identification Process

Wiggerts et al describes three different scenarios for object identification [Wiggerts97]. One scenario, function-driven, uses legacy functionality (subsystems performing a certain task) as its primary basis for class extraction. Another scenario, data-driven, first searches for persistent data elements, which often describe business entities, as the basis for its class extraction. The third scenario, object oriented, does not begin from the legacy system but begins by building an object model of the application domain. The thesis algorithm is a combination of those control and data flow scenarios.

In his work with reengineering COBOL legacy systems, Tsai (Joiner, 1998) concludes that because of incremental maintenance over many years, many COBOL programs are very difficult and expensive to maintain yet because these legacy systems are critical to business operations, they can not simply be replaced. Consequently, Tsai identifies the need to re-engineer these legacy systems, often to an object-oriented system.

Tsai states that converting a legacy system to an object-oriented system is difficult because the programming style that was used when the legacy system was developed does not incorporate itself into the object-oriented paradigm well. Tsai identifies the use of global variables within COBOL as a major obstacle (Joiner, 1998).

A typical approach to reengineering of procedural COBOL code to an object-oriented architecture (Sneed, 1992) is to partition programs into abstract data types and reallocate procedural programs according to their objects of processing. The result is a set of classes in object-oriented COBOL. Apparently, just restructuring procedural programs into classes has limited application in a large system because it leaves unresolved issues include such as how to produce non-redundant classes and how to make these newly-obtained classes in the new system communicate and collaborate with each other.

Levey (Levey, 1996) illustrated how older methods of writing COBOL programs lead to logic that clogs programs with complexity and chokes off further development; by using objects, Levey describes methodology for rebuilding theses systems. In Levey's process, the programmer gradually removes the older logic and replaces it with object oriented code. The result of this process are systems that are adaptable to new technologies in COBOL, although COBOL may not be an object-oriented language. It seems that the described approach employs many manual operations, which may not be effective in dealing with medium to large system.

In (Yang, 1994), a clustering/grouping method that is used to identify objects and the algorithms is implemented in the Maintainer's Assistant, a semi-automatic reverse engineering tool. Grouping can start with either a variable or a program statement that uses a variable, and then program transformations are employed to "regroup" a closure of variables and operations on these variables to form an "object". This technique is only an early step towards migrating a data intensive program into an object oriented program.

Newcomb and Kotik (Newcomb, 1995) use an object identification methodology which takes all level 01 COBOL records as a starting point for classes. They then continue to map similar records to single classes, and find sections of the COBOL program

that can be associated as methods to these records. This methodology can be highly automated and produces an object-oriented program that strongly resembles the original COBOL source program.

However as often happens during the long maintenance cycle of a COBOL program, many record fields are included in a record through incremental development which, had this program be designed from scratch, would have been put in a different record altogether. Certainly, much COBOL source code suffers from overall maintenance fatigue – jumbled code, record fields being included in huge records which should have been split into several smaller records, etc. Copying this illogical structure to an object-oriented program might not necessarily be in the best interests of reengineering.

(van Deursen, 1999a) uses both cluster and concept analysis for object identification, combining legacy data structures with legacy functionality. This approach, as in (Yang, 1994), takes the view that records are a natural starting point, and that decomposing the records into smaller ones is necessary, and a method of doing so was proposed. Concept analysis can help with problems that clustering analysis cannot solve, such as placing one item in different partitions and missing out useful items from partitions. Still, the results of these analysis results lack the information needed for building UML diagrams.

2.6.3 Background of Independent Task Evaluation

Tsai states that converting a legacy system to an object-oriented system is difficult because the programming style that was used when the legacy system was developed does not incorporate itself into the object-oriented paradigm well. Tsai identifies the use of global variables within COBOL as a major obstacle (Joiner, 1998). Another obstacle that that has been identified, and which is addressed in this paper, is the assumption of purely sequential processing in these legacy systems. Consequently, no independent tasks have been identified within this legacy system, and this makes it difficult to convert to a multi-processing, event-driven, object-oriented paradigm.

This method of using procedures as the maximum granular unit is similar to Gall's identification of procedures as incorporating the highest level of system control (Gall, 1998). A static call analysis was used on the selected legacy system in order to identify dependencies between procedures (Eisenbarth, 2001).

Lam, in her paper, tracks the amount of parallelism and misprediction of branching that is inherent in various programs of various types, such as numeric or database programs. The detected degree of parallelism was discovered to depend, in part, on the type of programs. Database and numerical programs tend to have a higher degree of parallelism than business-oriented programs, such as the selected legacy system (Lam, 1992).

2.6.4 Background of Event Identification

Kurfess (Kurfess, 1997) created a reengineering toolset which shares several components in common with ours including the translating of legacy source code to an intermediate representation; the transformation of this intermediate representation to an object oriented topology, and the parallelising of the reengineered system on parallel machines. The thesis' toolset has an additional feature in that it transforms the procedural-driven legacy system in its intermediate representation to an event-driven system through an event-driven system.

Pidaparthi (Pidaparthi, 1998) proposes a template to transform a procedural system into an object-oriented system through program transformations. Similarly, Cysewiski (Cysewiski, 1997) conducted a case study migrating procedural-driven legacy systems to object-oriented systems. However, both Pidaparthi and Cysewski do not specifically address the issue of transforming a procedural-driven system into an event-driven system.

Systa describes a process of extracting state diagrams from legacy systems by first creating an event trace diagram that models the interaction of a set of objects and actors during a specific usage of a system and then uses this event trace diagram to create a state diagram. Given a comprehensive set of test cases that provides coverage of all possible behaviours of a system, a legacy system is run using these test cases as input. The running system is monitored for the event and condition sequences that are produced by objects of that system. This event/condition sequence is sent to a tool, SCED, which constructs a scenario diagram that model the interactions of a set of objects implied by the event/condition sequence. SCED then is used to synthesise the general behaviour of an object as a state diagram, given a set of scenario diagrams in which the object participates (Systa, 1997; Booch, 1999).

Often the dynamic part of a system that is being reengineered is ignored in favour of the structural or static aspect. Because the functional or dynamic models affect the static model, it is important to incorporate these two models in order to understand fully the system. Pernul's (Pernul, 1995) dynamic model shows the life cycle of an object, which describes the update and state change behaviour of the object. Events are modelled as method invocations among objects. The state changes on objects caused by these method invocations are shown in the object lifecycle. The thesis' event identification algorithm identifies method invocations not only among objects in the legacy system, but between objects in the legacy system and external objects, such as File and System objects, as well.

2.7 Summary

As critical business systems aged, with incremental maintenance obsfuscating their original design and making it more difficult to adapt these systems to meet new business needs and to reduce their maintenance costs, techniques such as reverse engineering and object orientation were adopted as a solution.

Part of reverse engineering deals with program comprehension – the understanding of the components and functionality of the system - in order to enable the developers to make required functionality changes. One method used in program comprehension is function abstraction that ensures a uniform, consistent method of extracting the various tasks of a program. Program comprehension may also involve the extraction of business rules from legacy code in order to help stakeholders, such as managers, review the rules embodied in the system code to determine if their rules still meet current requirements. Object orientation, through modularisation of a program, promises a technique to reduce a program's complexity and decrease its maintenance costs.

Chapter 3: Related Work

3.1 Introduction

In this chapter, several topics are introduced that will be used within or have relevance to this thesis as will be seen in later chapters. A brief introduction to formal methods is provided along with why WSL, with its advantages, was selected over other possible formal notations. Program understanding, along with software visualisation and domain knowledge is introduced. Domain engineering and knowledge is introduced in order to both explain its role in program understanding and the need to abstract program specifications from the source code. An introduction to the various UML diagrams is provided. The requirement for program understanding, and consequently software visualisation, is explained in order to enable developers, among other system stakeholders, to more fully understand the system in order to adapt it to new requirements.

3.2 Formal Methods in Reverse Engineering

In life-critical systems, it is important that developers employ methods that offer a high degree of assurance that a system's design accurately captures the system's critical requirements and that the implementation of the system in software is an accurate realisation of the system's design. Formal specification and verification techniques were developed in order to provide this degree of software assurance in the software development process (Kemmerer, 1990). Formal methods have been proposed as a method to enable software to be constructed so that it operates reliably regardless of its complexity. Formal methods can be defined as mathematically based languages, techniques, and tools for specifying and verifying such systems. Although formal methods does not guarantee *a priori* correctness, formal methods can increase our understanding of a system by revealing inconsistencies, ambiguities, and incompleteness that might go undetected. (Wing, 1996).

In life-critical systems, it is important that developers employ methods that offer a high degree of assurance that a system's design accurately captures the system's critical requirements and that the implementation of the system in software is an accurate realisation of the system's design. Formal specification and verification techniques were developed in order to provide this degree of software assurance in the software development process. However, many of these techniques traditionally have been an after the fact implementation where the system had been developed using standard, non-formal software development techniques and then the original specifications were translated into a formal specification with the properties of the software program formally verified to fit this formal specification (Kemmerer, 1990).

Formal specification expresses the user requirements, traditionally written in natural languages, into a precise mathematical notation which describes the system and its desired properties. This formal specification can be refined into a system design and then verified to ensure that the design matches the specification (Wing, 1996). Kemmerer argues that formal verification at the code level must occur in order to ensure the accuracy of the design (Kemmerer, 1990). A tangible by-product of this specification is an artefact, which can be formally analysed for consistency, that provides a higher level of abstraction than the system source code (Wing, 1996).

High-level specifications of the system are defined in a precise mathematical notation that describes the system's behaviour while omitting many implementation details. These high-level specifications are further refined into several, less abstract but more detailed specifications, also expressed in a precise mathematical notation. The specifications must be proven to be consistent with the critical requirements by using either an algebraic or a state machine approach. In the state machine approach, the effect of performing each operation is based on certain conditions being satisfied when the operation is invoked; the formal critical requirements must be verified to satisfy the initial state of the state machine and then, every operation preserves these critical requirements. In this way, the design, as represented by the specifications, can be proven to satisfy the formal model (Kemmerer, 1990).

Common COBOL control constructs, such as PERFORM, have been defined into a formal semantics by Baumann. Baumann states that this formal semantic basis of programming language constructs is necessary in order to perform correct program analysis and semantic invariant program transformations. These transformations include abstract interpretation and partial evaluation (Baumann, 1993; Jones, 1993; Mosses, 1991).

Baumann states that reverse engineering methods must be based on a sound foundation, which entails formal denotation semantics, because if these methods should extract the wrong information during reverse engineering process, this wrong information could lead to new errors in the reengineered programs. Although many reverse engineering methods do not involve formal methods, these formal methods, through the their ability of consistency checking, have the potential to avoid the occurrence of ambiguous programming constructs such as in Algol 60 (Baumann, 1994).

Some formal methods, such as Z or Larch, focus on specifying the behaviour of sequential systems. States are described through sets, relations, and functions with state transitions described in terms of pre and post conditions. Some languages, such as Harel's statecharts, focus on specifying the behaviour of concurrent systems. Regardless of the formal method, all of these formal methods use the mathematical concepts of abstraction and of composition (Wing, 1996).

In the wide spectrum language approach, the specifications and the program can be expressed in the same language. Typically, specification and program constructs are intermixed. In this way, high-level specifications are gradually refined into programs. The wide spectrum language approach offers several advantages: refinement of specification entails refinement of the program as well; the same modularisation constructs can be used to structure specifications and programs; and there is a gradual transition rather than a sudden leap from high-level specifications to efficient programs (Sannella, 1993).

Z is a specification language based on the idea that both programs and data can be denoted using set theory. Z models data types using set-theoretic constructions. Z can specify pre and post conditions of functions (Sannella, 1993). Paige argues that because Z is not a wide spectrum language, it is much more difficult to conduct refinements of programs specified in Z (Paige, 1999).

In order to be successful in reverse engineering, a formal description technique must fulfil at least three conditions:

- This technique must be applicable to parts of the programming language that is relevant for reverse engineering methods.
- The technique should support standard approaches to program analysis such as control-flow analysis, data-flow analysis, etc.
- The technique should be easily and efficiently implementable (Bauman, 1994).

WSL fulfils these criteria in that in this thesis, WSL is extended to represent the constructs of the source code in question, adapts to program analysis techniques such as control flow analysis as demonstrated in the UML diagrams, and WSL is incorporated into various reengineering tools for an easy implementation.

By providing UML with a formal description via a formal intermediate language representation, several advantages can be achieved:

a) clarity – unambiguous semantics

b) consistency – ensure consistency between different components

c) extendibility – verify the extensions of the UML as to its soundness

d) refinement – to allow design patterns to be checked for correctness

e) proof – to allow justified proofs and checks of important properties of a system, such as safety and liveness properties (Clark, 1997)

Some of the disadvantages of formal methods are that it often entails a steep learning curve for developers and implementing an abstract model, in formal notation, of a system and keeping this model synchronised with any system changes is difficult and time consuming. One example, a GPS system took two staff months to model and to formalise the system requirements in PVS (Crow, 1998). Although some research has been done to automatically extract some UML diagrams from C code (Musuvathi, 2002), no existing tool extracts a set of UML diagrams from WSL, or from a formal intermediate representation, regardless of the original programming language source code. By incorporating an industry standard modelling notation, UML, the developer's learning curve to understand the extracted model is greatly reduced; furthermore, this tool can be run after system changes in order to keep the generated model, represented as a series of UML diagrams, synchronised with a changing system.

3.2.1 WSL

3.2.1.1 Background of WSL

A computer program can be described as a function that translates an input state to an output state. A specification can be termed to be a description of this function. A program, A, can be refined by another program, B, in that the set of possible final states for the implementation, B, is a subset of final states for the program, A. This refinement also results in the outputs of program A and B being identical and, hence, equivalent even though the programs themselves are quite different structurally (Yang, 2003: pp 3-4).

If the specifications are written in a formally defined mathematical language, it is possible to prove that a given program is a correct implementation of a given specification. Usually, this process involves a number of intermediate stages where a formally written specification is further developed into a program. In order to reduce the number of these intermediate stages to a minimum, specifications should be included as part of the programming language. If the language includes low-level programming constructs, then this language can be termed as the wide spectrum language (WSL). The wide spectrum language is so termed because it embraces the whole spectrum from mathematical specifications to executable implementations (Yang, 2003: p 4). WSL contains both specification constructs, such as the general assignment statement, and programming constructs, such as while-do loops (Younger, 1993).

Because WSL represents specifications and executable implementations, it is ideal for reengineering and re-documentation purposes. By translating a legacy system's source code to WSL as an intermediate representation, a reengineering tool can derive both the specifications inherent within the WSL representation in order to generate the documentation of the system through a series of UML diagrams (Yang, 2003: p 4).

Intermediate languages or representations in reengineering are very commonly used in many commercial modelling tools. The use of intermediate languages or representations has many advantages including the ability to use standardised transformations and mappings from the intermediate to the target domain and, thus, avoid the "impedance mismatching" problem between the source and target domain. The use of intermediate representations allows the reengineering effort to be divided up into smaller steps rather than as a monolithic source to target domain reengineering effort (Millicev, 2002).

3.2.1.2 Advantages of WSL

WSL was developed with several advantages in mind:

- The ability to express general specifications in terms of mathematical logic with suitable notation. This ability is found in WSL and Z.
- In order to avoid requiring a developer to express everything about the program being reengineered, some executions can be specified that will not result in only one particular outcome but in one of a permitted range of outcomes.
- A well-developed library of proven transformations that do not require the user to fulfil complex proof obligations before these transformations can be applied.
- Techniques to bridge the "abstraction gap" between specifications and programs. The need to resolve this "abstraction gap" is important in this reengineering effort where a legacy system is re-documented and re-programmed in another programming language.
- Applicability to real programs not just a set of limited programs written in languages. This reengineering effort concentrates on a real world legacy system written in the COBOL programming language.
- The ability to scale to large programs. The selected target system is a large (112 000 lines of COBOL code) system (Yang, 2003: pp 5-6).
- WSL has existing tool support as well. The FERMAT tool was designed to use WSL and has applications in the following areas:

1) Improving the maintainability of existing mission-critical software.
2) Translating programs into modern programming languages. Fermat often translates program written in obsolete assembler language to more modern languages such as C.
3) Extracting reusable components from the current system, deriving their specifications, and storing the specifications, implementation, and development strategy
4) Reverse engineering existing systems to high-level specifications, followed by subsequent re-engineering and evolutionary development.

Other intermediate languages other than WSL were examined as to their suitability for this reengineering project. Among them were the process algebra family of CCS (Milner, 1980), TCSP (Hoare, 1985), and ACP (Begrstra, 1984). Process algebra can be described as a set of action (or event) symbols, a set of operations, and a set of axioms that describe the operator's properties. Individual processes can be represented as algebraic expressions involving sequencing and choice among actions. Some advantages of process algebra are that it can represent parallel processes through a parallel composition operator and that it allows the use of axioms to assert equivalencies among expressions. The latter advantage allows complex expressions to be simplified and demonstrates that a given expression satisfies a selected specification (Yeh, 1991). Process algebra is well suited to describe packet switching processes through protocol specification languages such as LOTOS (Brinksma, 1986) and, in the intermediate language ACP, represents finite-state asynchronous systems with the ability to hide details of subsystems (Bergstra, 1984). Although process algebra languages are popular due to their relative ease of manipulation and rich abstraction capabilities (Yeh, 1991), WSL was adopted because, among many reasons, its ability for proven transformations and the ability to represent high and low levels of abstraction which make it well suited for reverse engineering purposes

3.2.1.3 Typelessness of WSL

Because WSL lacks data typing, it may, at first glance, present many problems but, after further analysis, reveals that typelessness is not such a disadvantage as it first seems. An example, the conversion from COBOL to C++ via the intermediate representation WSL, which is outside the scope of this thesis but which this tool has the nascent ability to perform, presents many problems in the conversion. The problems for this conversion lie, not in the typelessness of WSL, but in the strong data typing and weak data sizing nature of C++ compared to the weak data typing but precise data sizing of COBOL. An example, in COBOL, you can transfer one record consisting of numeric and string fields to another record, of the same size, consisting of purely alphanumeric characters. Record fields within COBOL may be accessed by their index from the base of the record rather than record name. COBOL will automatically convert an integer stored as a string in a record to an integer. In C++, record fields normally are accessed by record name. In C++, there is no automated conversion of a string to an integer. Furthermore, in COBOL, a numeric variable may be specified precisely in terms of the exact size of the base and mantissa in the case of floating point numbers. In C++, floating-point numbers are stored in set memory sizes. If a floating number uses less space than allowed, this space is wasted. If the number exceeds the set size of its type, there is no way to programmatically expand this variable's size other than declare it of a different, longer data type. Conversion of data types from one programming language to another often involves data type conversion, remodeling of code that handle the variables of these data types, and the conversion of existing data (Feingold, 1981; Lippman, 1998).

Furthermore, the typelessness of WSL is not necessarily a disadvantage when expressing an abstract modeling notation, such as UML, using WSL. Many programming languages lack the typing that UML uses to express abstract modeling elements or their

attributes, such as classes, messages, or asynchronicity. Consequently, it is difficult to find an intermediate language that has the typing needed to express abstract models such as UML. Furthermore, the typing of this language must be changed every time that the UML specification changes and new modeling elements are introduced or modified.

Although UML has primitive data types such as Boolean or string, these primitive data types are either not present in many programming languages or vary in the way that they are represented and manipulated. An example, in Visual Basic, a developer does not have to express a string of a set size. In C++, the developer must define this string of a set size. If the program exceeds this set string size, the program produces a run-time error. The representation and manipulation of these data types have relevance in the way that UML will model its diagrams. For example, in the C++ program, the "run-time error of out of bounds string size" must be modeled whereas in Visual Basic, since this event does not normally occur if at all, this event need not be modeled (Brown, 200; Lippman, 1998).

3.2.1.4 WSL Kernel Language

The WSL language is built up in a series of layers from a small mathematically tractable "kernel language". This kernel language consists of the following primitive statements:

Assertion {P} - if formula P is true, the statement terminates; otherwise, it aborts or never terminates

Guard [Q] – restricts previous non-determinism to statements whose condition for execution requires Q

Add variables: add (X) adds the variables in X to the state space

Remove variables: remove(X) removes the variables in X from the state space

Sequence (S1:S2) executes S1 followed by S2

Non-deterministic choice: chooses one of S1 or S2 for execution, the choice being made non-deterministically

Recursion – a statement, S1, may contain occurrences of X as one or more of its component statements. These statements represent recursive calls to the procedure whose body is S1 (Yang, 2003: pp 7-8).

The use of guards enables the flow of execution to be made deterministically. In this way, the use of guards, plus encapsulated statements upon whose execution is dependent on these guards' evaluation of their condition, approximate the **if** statements of a programming language (Yang, 2003: p 9).

3.2.1.5 Extensions to Kernel Language

By building on the basic constructs of the WSL language, extensions to the WSL language include simple assignment, deterministic iteration (do-while loops) and deterministic choice (if-then) statements along with blocks of statements within procedure calls (Yang, 2003: pp 21-24).

In this thesis, the WSL kernel language was extended to include such constructs as class structure that contains both variables (attributes) and procedures (methods or operations).

The **Var** construct is extended to include structures of both record and *Class* type. A record structure is defined by using the **Var Struct** keywords in the following format:

Var Struct RecordName

Begin /* beginning of record fields */

Var VarName1

Var VarName2

End /* end of record fields */

The class structure is defined by using the *Class* keyword in the following format:

Class ClassName

Begin /*beginning of class structure */

Var VarName1 /* Attributes of Class */

Var VarName2

Proc ProcName2 /* Methods of Class */

Begin

/* WSL statements */

End.

End

To access a record field enclosed within a record data structure, the following format is used:

RecordName.FieldName

To access a class attribute or method enclosed within a class structure, the following format is used:

ClassName.AttributeName

In order to represent an array of elements within one variable, the concept of an array is used. The format of the array variable declaration is as follows: **VAR** <*Variable_Name*>**[**<*Max_Element_Size*>**]**. Elements within the array are accessed by their element index, in the form <*Array_Variable_Name*>**[***Element_Index***]** where *Element_Index* represents the index of this array.

3.2.1.6 Tools Based on WSL

Once a source program has been translated into WSL, the WSL representation may then be inputted to such tools as Maintainer's Assistant (Bull, 1995) that has the capability for program transformations, code reductions, etc of this WSL representation. Maintainer's Assistant has evolved into an industrial-strength re-engineering tool, FermaT. The FermaT transformation system (Ward, 2001) allows transformations and code simplification to be carried out automatically. It has the capability of enabling proof-of-correctness testing (Millham, 2003). However, because Fermat lacks the ability to restructure code and to provide abstract views of the system, in industry-standard UML, Fermat was judged to be unsuitable for the thesis' reengineering purposes.

3.3 Program Understanding

Program understanding has the ultimate goal of enable the comprehension of the underlying functional and data concepts of the program. Program understanding begins by analysing the syntactical units of a given source code which are delivered by code browsing and developing a cross reference analysis of singular items grouped into variables, reserved words, strings, constants, etc. The internal representation of the program is represented via an abstract syntax tree. Knowledge of syntactic structures is necessary in order to segment a monolithic program into coherent, logically organised modules (Rugaber, 1993).

This knowledge representation may be in multiple formats such as textual or graphical notation. Graphical notation has the advantage over textual formats in that graphical notations can more clearly depict complex relationships between model elements, such as class (Rugaber, 1993).

Rumbaugh et al identifies three viewpoints necessary for understanding a software system: the objects manipulated by the computation (the data model), the manipulations themselves (functional model), and how the manipulations are organised and synchronised (the dynamic model) (Rumbaugh, 1991).

Rugaber states that most of the representation techniques, outlined below, emphasise one of these views.

Representation	References
Object-Oriented Frameworks	(Johnson, 1988)
Category Theory	(Srinivas, 1991)
Concept Hierarchies	(Biggerstaff, 1989; Lubara, 1991)
Mini-languages	(Neighbors, 1984; Arango, 1986)
Database languages	(Chen, 1986)
Narrow spectrum languages	(Webster, 1987)
Wide Spectrum languages	(Wile, 1987; Ward, 1989)
Knowledge Representation	(Barstow, 1985)

Table 3.3.1 Table of Representation Techniques

UML, through its various UML diagrams, encompass all of Rumbaugh's views. The data model is represented by UML's class and object diagrams. The functional model is represented by activity and state diagrams. The dynamic model is represented by sequence and collaboration diagrams.

In order to understand a program, a semantic analysis of the program, that reveals information about the control and data flow of the program, is needed. The control flow links that mirror the sequencing of the program and the calling links are analysed. Control dependency analysis checks the reachability of code parts of the program depending on the evaluation of conditional statements. The data structure analysis is important in order to understand all the used data and the interrelationships concerning data dependencies (Baumann, 1993).

Although dynamic data mappings are highly desirable, these mapping require an exhaustive semantic analysis that often exceeds affordable constraints on reengineering time and complexity. Knowledge-based techniques are used to extract the functional concepts contained within programs (Baumann, 1993).

In order to modify a given program for new functional requirements, one performs a stepwise refinement of understanding of the program. One needs to first understand which parts and what type of changes are necessary to implement these changes in the program. This usually involves a type of simple syntactic analysis of the existing program followed by a more complex semantic analysis to detect all the necessary global and local interrelationships that may affect the required changes in the program (Baumann, 1993).

In comparison with the 60 grammar rules used to express the grammar of C, the grammar of COBOL 74, one dialect, consists of more than 170 complex production rules. This huge number of rules makes developing a complete formal semantics for COBOL impractical (Baumann, 1993).

Rugaber sees the structure of the source code as embodying certain design decisions. It is important to extract this information in order to try and understand how the original designers arrived at the solution that they did. An example, procedures and functions are a way of generalisation because these functions represent a generalisation of a specific function within a program. Encapsulation, the modularisation of functions and variables within clusters, and the definition of clear boundaries, or interfaces, between these clusters, is useful to program comprehension in that the encapsulated construct might be considered by a specification that is much smaller than the amount of code in the component. Furthermore, if the component hides the details of a major design decision, side effects of the change are limited when that decision is altered during later maintenance (Rugaber, 1990).

3.4 Reasons for Necessity of System Visualisation in Reverse Engineering

Program understanding can be defined as the process of developing an accurate mental model of a software system's intended architecture, purpose, and behaviour. This model is developed through the use of pattern matching, visualisation, and knowledge-based techniques. The field of program understanding involves fundamental issues of code representation, structural system representation, data and control flow, quality and complexity metrics, localisation of algorithms and plans, identification of abstract data types and generic operations, and multiple view system analysis using visualisation and analysis tools. Reverse engineering involves analysing data bindings, common reference analysis, similarity analysis, and subsystem and redundancy analysis (Whitney, 1995).

Program understanding involves the use of tools that perform complex reasoning about how behaviours and properties arise from particular combinations of language primitives within the program. One method of program understanding is to use visitors, or small reusable classes, whose function is to parse the source system and evaluate the combinations of language primitives that had been discovered during parsing (Wills, 1993). Other methods try to evaluate and understand a system by taking, as input, the

goals and purpose of the system as a specification (Johnson, 1986). Another method is to use clichés which try to recognise commonly used data structures and algorithms and then match these structures and algorithms to higher level abstractions. Examples of clichés are the structures and algorithms associated with hash tables and priority queues. The degree of accuracy of the matching varies with the goal of program understanding. An example, an exact match is needed for program verification while only a reasonably close match is needed for documentation purposes (Harris, 1995).

Software visualisation is a technique to enable humans to use their brain to make analogies and to link a visual software representation with the ideas that this representation portrays. This link would be much more difficult to make if the software representations were purely in textual form. Software visualisation relies on techniques such as animation, graphic design, and cinematography (Price, 1992).

Software visualisation has been used for decades in order to help developers understand programs. These techniques vary from flowcharts (Goldstein, 1947) to animated graphical representations of data structures (Baecker, 1981).

Program visualisation systems or tools can be characterised according to their scope, content, form, method, interaction, and effectiveness. Scope refers to the visualisation system's general characteristics such as whether it models concurrent programs or whether there are any size limitations as to the system being depicted. Content refers to the content being visualised. Some visualisation systems can model both data and code while others model algorithms only. Form refers to what elements are being used in the visualisation. Some visualisation systems use animated graphics while other systems provide multiple views of different parts of the system. Method refers to how the tool specifies the visualisation. Does the tool require the program source code to be modified in order for it to be visualised? Some tools require the user to insert special statements in code of special interest in order for this code to be properly visualised. Interaction refers to how the user interacts and controls the visualisation. How does the user navigate through the visualisation of a large system in order to see how different parts of the system are being modelled? Effectiveness refers to how well the visualisation communicates information regarding the system being visualised (Price, 1992).

Creating a model of a system is a useful way to understand the system for many reasons. One reason is that the system itself is a model of an external business process and as such, this model exists, thus reducing the problems in creating further models. Another reason is that programs are constructed from executable and data structures – these existing entities can be extracted and analysed in order to produce a model of the structure and dynamics of the system (Hall, 1992; Rugaber, 1993).

Rugaber identifies the understanding of a software system by constructing a mental model of the system as an important component of any task involving a software system. Much work in reverse engineering is devoted to explicit knowledge representation of a software system (Van, 1992; Rajlich, 1992). This explicit knowledge representation results in a set of well-defined system models, invaluable to those undertaking any task on the software system, and better prediction of the reverse engineering's expected results.

3.5 Unreliability of Source Code for Documentation Purposes

Use source code as the sole basis for obtaining an understanding of the system has many disadvantages, particularly in legacy systems. In many legacy systems, the original design of the system has been obfuscated by the many incremental changes during the system's long maintenance history. Furthermore, the end-users, whose requirements originally help design the system to meet their business needs, have usually long gone and the documentation outlining these requirements are often missing. Without these original end-users and no documentation, it is often difficult to determine the exact business processes that these systems model (Kwiatkowski, 1998).

Source code is very programming-language-dependent (Yang, 1999). In order to understand the code, the developer must be fully proficient in the programming language used to develop the system. The function and role of each section of source code within the system may be obvious to a developer but may be meaningless to a non-technical end-user. End-users want to see how the business processes that the system represents are modelled and they want to ensure that all of their business requirements are met in the system. End-users are not concerned with the internal design and details of this system.

It is difficult for developers to view parallel data and control flows from reading the code. The control logic, especially if this control logic is heavily nested, is difficult to visualise from the source code, particularly in order to quickly identify which control constructs affect which parts of code. It is difficult to visualise events occurring in various parts of code and to visualise how these events interact with various objects in the system. Relying on source code as the only documentation source makes it difficult to view the interaction of system objects with other objects and external actors.

Source code encompasses many perspectives (such as objects, deployment of components, and timing of object interactions) within itself. These multiple perspectives are confusing – it is difficult to represent the source code in each separate perspective. Source code has the additional disadvantage in that it is difficult to represent abstract concepts and behaviour from low-level, detailed source code.

3.6 Domain Knowledge and Reengineering

Domain knowledge engineering is the development and evolution of program boundary specific knowledge and artefacts to support the development and evolution of systems in the domain. Domain engineering takes advantage of the similarities inherent in systems within one domain and builds a domain model that defines the basic structure for existing and future requirements of systems. (Howe, 2002). In terms of legacy systems, domain engineering concentrates on the way in which the legacy system perform its main tasks in the original operation environment and the new environment, the way in which the operational environment affect its performance in its original and new context, and the way in which the legacy system is constructed to adapt to different usage contexts (Li, 2001).

Because the legacy system is large, unstructured, complicated, and old-fashioned, it is difficult to read and understand. Although the source code has been improved, it is still unreadable. The specification from the source code is needs to be realised, even if the specification differs from the present source code (Breuer, 1991). Because no method of code comprehension is perfect and it relies on the involvement of the human beings completely, it is important for the behavioural aspect to be understood for the changes and enhancements of the legacy source code (Yang, 1997b). Behavioural specifications consist of a set of top-level behaviour definitions, an explicit definition of the valid domain, and a set of lower-level local specifications. The specifications of the legacy code are generated entirely with a line-by-line translation mechanism (Bergey, 1997). Therefore, the next step is to present the specifications in graphical manner, which means extracting the high-level representation that is more understandable than the textual specifications within legacy code. One method to graphical represent this legacy system's specification is with UML activity diagrams, which describe the behavioural aspect of the modelled legacy system (Breuer, 1991). Another method is to use an object abstraction technique, in the case that a section of code implements a mathematical function or a simpler description of a higher-level behaviour or auxiliary behaviour, so that a more abstract specification of this functionality as well as more detailed one is provided (Gold, 1998).

3.7 UML

One of the most common graphical notations is UML (Unified Modelling Language). UML provides multiple perspectives of the system. Use case diagrams model the business processes embodied in the system from a user's perspective. Statecharts and class diagrams model the behaviour and structure of the system respectively; the behaviour and structure of the system would be of most interest to developers.

This section provides a brief description of the various UML diagrams, along with the aspects of a legacy system that each diagram type models. The advantages of UML as a modelling language are outlined.

One of the most common graphical notations is UML (Unified Modelling Language). UML provides multiple perspectives of the system. Use case diagrams model the business processes embodied in the system from a user's perspective. Statecharts and class diagrams model the behaviour and structure of the system respectively; the behaviour and structure of the system would be of most interest to developers (Muller, 2000a).

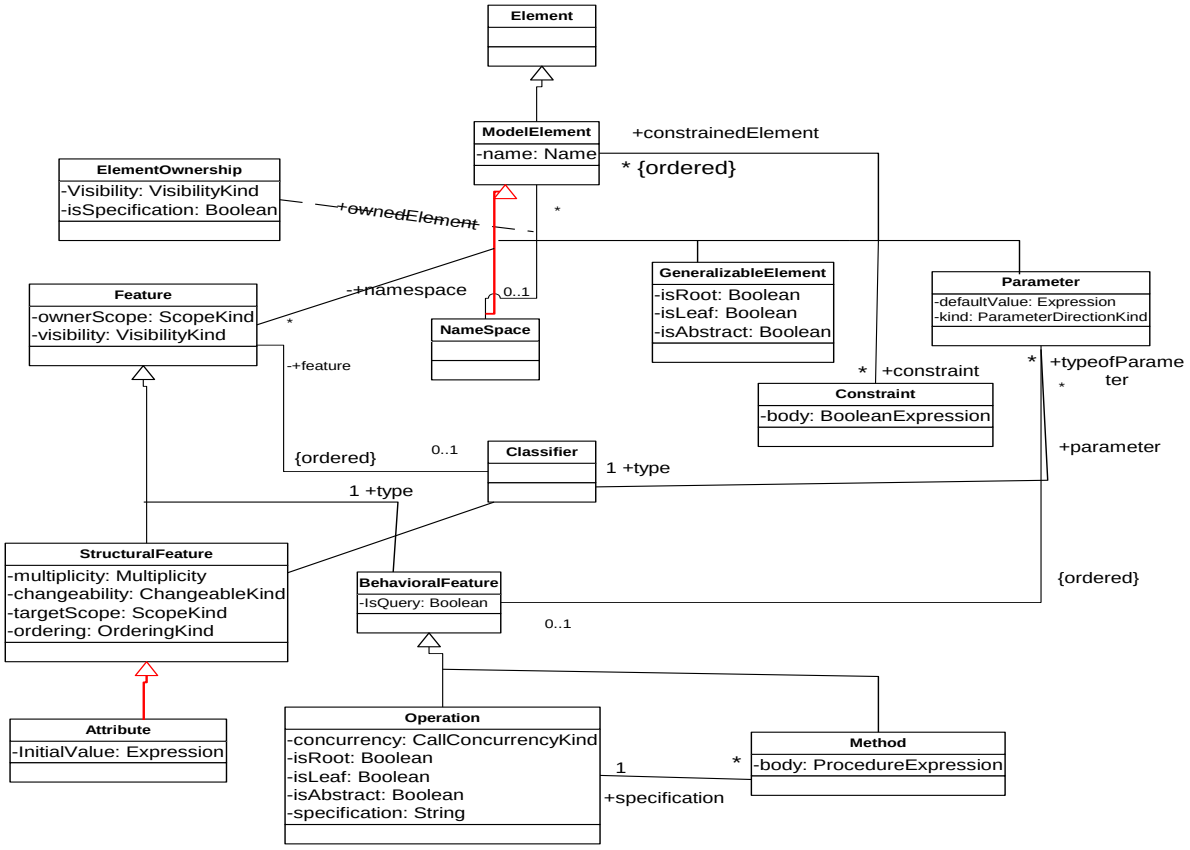


Fig. 3.7.1.1 UML Core Package (OMG, 2004: p 2-13).

The UML Core Package that provides the semantic basis for modelling the modelling elements used in depicting the various behavioural and structural diagrams of the UML language. This Core Package provides the core constructs required for the basic metamodel of the UML language as well as providing an extension mechanism for the UML language through its ability to add metaclasses to the Core Package using generalisations and associations (OMG, 2004, p 2-12). This ability to extend the UML, as well provide the semantic basis for the core constructs used in modelling elements of the various UML diagrams, provides UML with an expressiveness, a unity, and the ability to extend (Muller, 2000a).

3.7.1 Diagrams of UML

UML has different types of diagrams: Use Case, Class, Object, Sequence, Collaboration, Statecharts, Activity, Component and Deployment (Muller, 2000a).

3.7.2 Use Case Diagrams

Use case diagrams emphasise services and operations that a system offers to entities outside the system. Use case diagrams are often used to model the business processes that the system represents (Muller, 2000a).

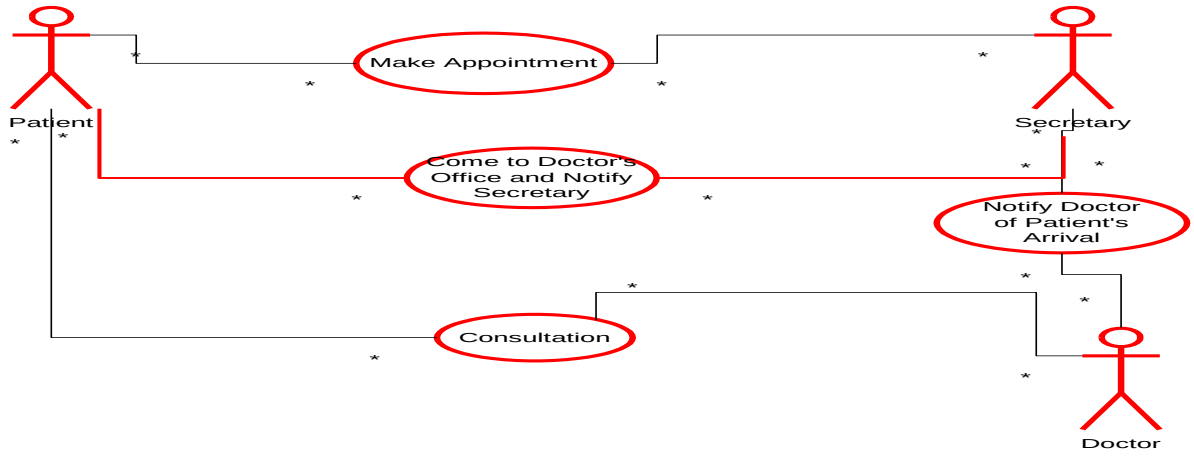


Fig. 3.7.2.1 Use Case of a Patient-Doctor Visit.

In this example, the business processes in seeing a doctor are modelled. A patient makes an appointment with the doctor's secretary and arrives at the appointed time, notifying the secretary of their arrival. The secretary will then notify the doctor whereupon a consultation between the patient and doctor occurs.

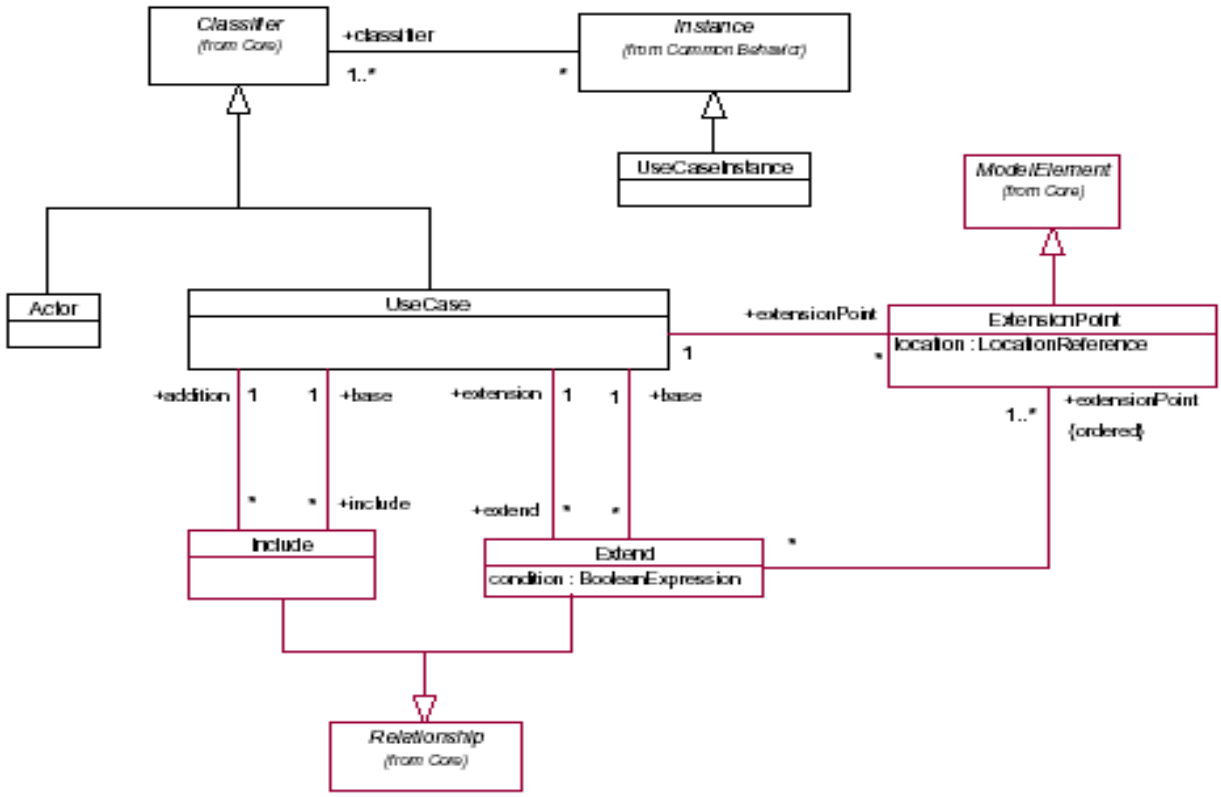


Fig. 3.7.2.2 Use Case Package (OMG, 2004: pp 2-130).

In Figure 3.7.2.2, Use Case Package, the relationship between the UML Core Package and the Use Case Package is demonstrated. A Use Case and Actor are of type Classifier from the UML Core Package. Relationships of include and extension allow the Use Case to include other user cases or to have its behaviour extended respectively. The Extend model element includes the attributes of Condition, which specifies the condition that must be fulfilled if the extension is to occur, and include the association of base (the Use Case to be extended) and Extension (the Use Case Specifying the Extending Behaviour). The Include association

includes the addition, the UseCase specifying the additional behaviour, and the base, the UseCase that is to include the addition (OMG, 2004: pp 2-130-2-132).

An actor is an external entity to the system and is used to define the set of roles that users of an entity can plan when interacting with the entity (OMG, 2004: p 2-131).

3.7.3 Class Diagrams

Class diagrams emphasise the static structure of the system. Object diagrams model the static structure of a system at a particular point in its execution. Object and class diagrams can be differentiated in that classes model the structure of a system while objects are specific examples of the structure (Muller, 2000a).

Fig. 3.7.3.1 One-to-many relationship between Employer and Employee Classes.

Here is an example of a class diagram and its association between two classes, the classes of Employer and Employee. This class diagram models a simple employer-employee relationship between two entities, employer-employee. The Employer class has attributes such as Company Name and performs operations such as doing the payroll for its employees. The Employee class has attributes such as the employee name and company number. The Employee also performs an operation such as their job function.

3.7.4 Component Diagrams

Component diagrams model software components and their relationships within the implementation environment. These components may be simple files or dynamic libraries. Relationships between components are modelled as dependency relationships; a relationship between two components is identified when one component offers services to other components. Generally, these components represent compilation dependencies. Several of these components may be grouped into packages, or subsystems, according to some logical criteria (Muller, 2000a).

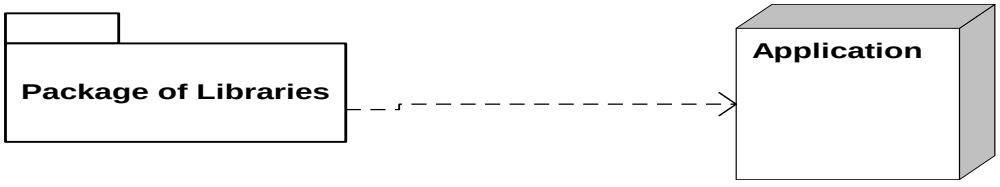


Fig. 3.7.4.1 Component Diagram of an Application and its linked package of libraries (Pu, 2005).

A simple component diagram that models the dependencies between an application and a package of code libraries upon which the application is dependent upon for its correct functioning is depicted in Fig. 3.7.4.1.

3.7.5 Deployment Diagrams

Deployment diagrams model the relationships between physical entities of an implementation environment (Muller, 2000a).

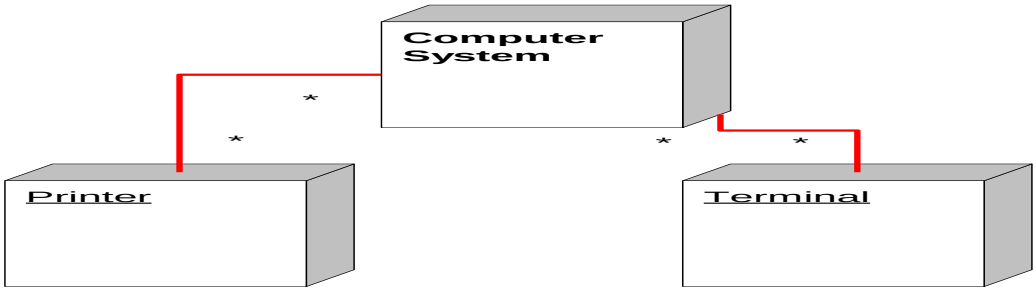


Fig. 3.7.5.1 Deployment Diagram of a Computer System with Associated Peripherals (Pu, 2005).

This figure highlights the relationship between a computer and two if its peripheral devices, a printer and a terminal.

3.7.6 Sequence Diagrams

A sequence diagram is an interaction diagram that details how operations are carried out -- what messages are sent and when. Sequence diagrams are organised according to time. The time progresses as you go down the page. The objects involved in the operation are listed from left to right according to when they take part in the message sequence (Muller, 2000a).

Sequence diagrams are modelled as one line; this line containing all the objects in the system with each object in its own swimlane or partition (Muller, 2000a).. Because all objects have a global scope (variables in COBOL are globally scoped), these objects are shown as being immediately activated after their declaration in the swimlane or partition. Messages are depicted with their origin at the source object and their endpoint being the target object (Muller, 2000a). These messages are depicted as synchronous or asynchronous.

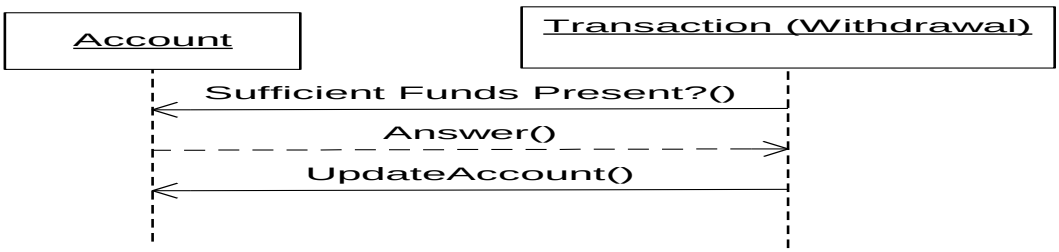


Fig. 3.7.6.1 Example of a Sequence Diagram Illustrating a Bank Transaction (Pu, 2005).

An automated teller machine handling a withdrawal transaction sends a message to the person's, who is requesting the transaction, bank account to see if sufficient funds are available to manage this transaction. The transaction object, within the automated teller machine, receives a reply. If yes, the transaction requests the Account object updates its account information to reflect the transaction. If no, then the account information is updated to reveal a failed transaction occurred.

3.7.7 Collaboration Diagrams

Collaboration diagrams model messages exchanged over time among the objects and their links within a system (Muller, 2000a).

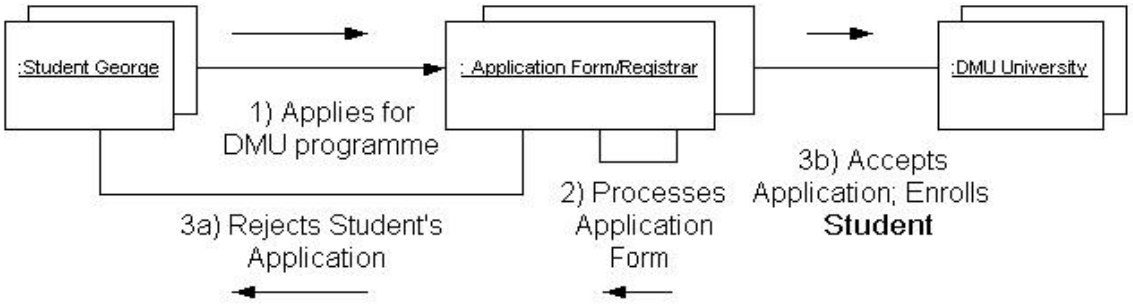


Fig. 3.7.7.1 Collaboration Diagram of a Student Applying to DMU University.

Collaboration diagrams are very similar to sequence diagrams. Both types of diagrams model the type and sequence of messages exchanged between objects. However, sequence diagrams emphasise the object communication while the collaboration diagrams emphasise the structure of the collaboration (Muller, 2000a).

A small scenario used as the basis of a collaboration diagram is given as George, a student, who meets the entry qualifications (as self-message) and then applies for a programme at DMU University via DMU's application form. DMU University, in turn, processes this application and notifies the applicant of the result (acceptance/rejection).

3.7.8 Statecharts

Statecharts model how an object changes or responds to external stimuli within a system. Statecharts model the changes of state embodied within a system due to messages. Activity diagrams model how an object changes or responds to internal processing within a society. Activity diagrams model the flow of control and of information within a system (Muller, 2000a).

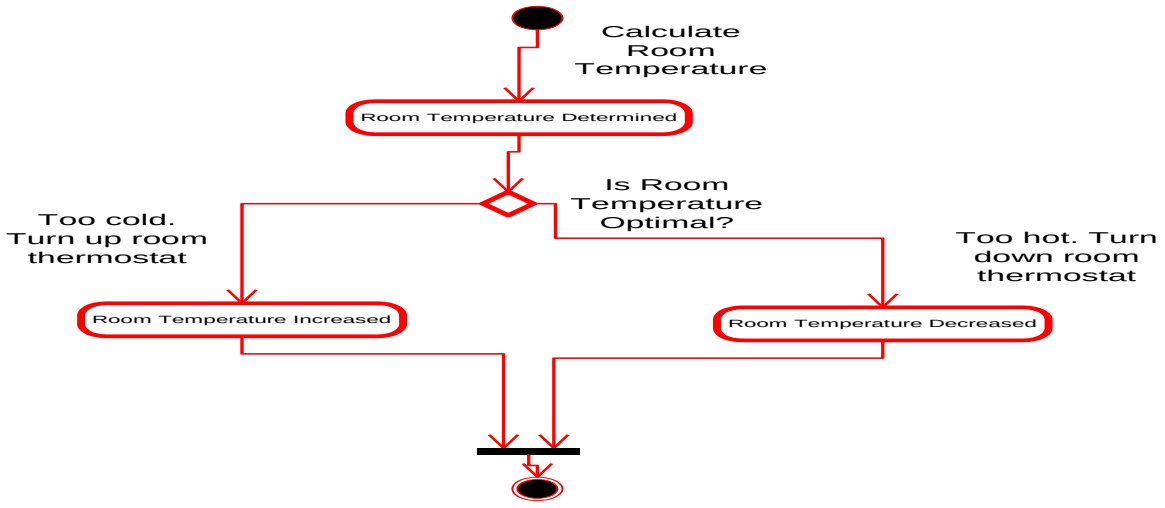


Fig. 3.7.8.1 Statechart of a Mechanism Maintaining Room Temperature.

Here is a simple statechart that shows the room temperature being determined and two courses of action being taken depending on the room temperature. If it is too cold, the thermostat is turned up; if too hot, it is turned down. This statechart shows the states, decisions, and transitions that would accompany such a simple activity.

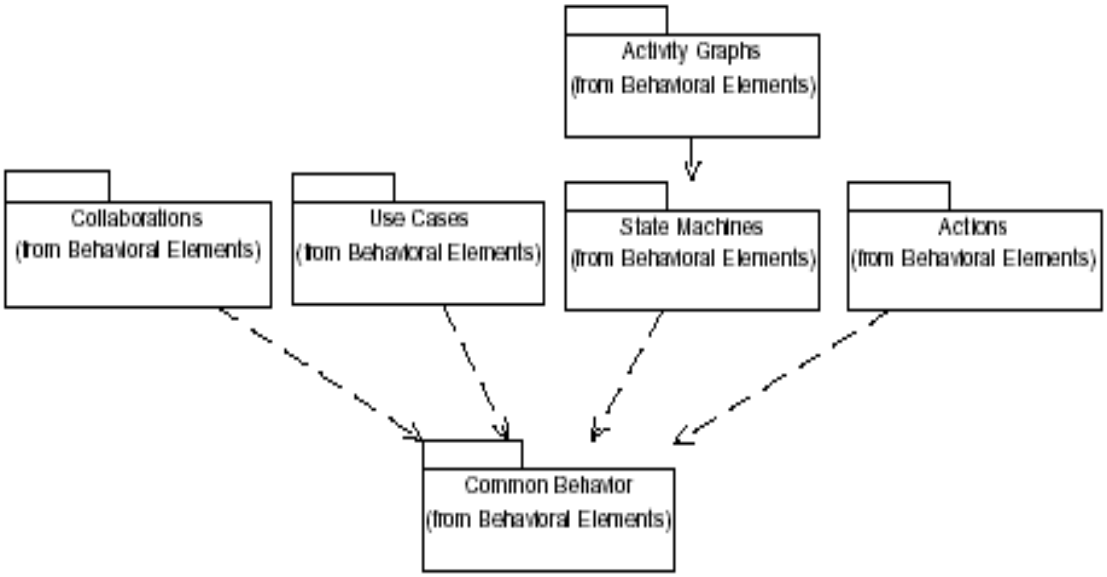


Fig. 3.7.8.2 Behavioural Elements Package (OMG, 2004: pp 2-7).

Various activity diagrams such as Use Cases, StateMachines, and Activity Diagrams are derived from a common Behavioural Elements Package. This Behavioural Elements package defines the core components used for the dynamic elements of UML and provides the infrastructure for the included behavioural diagrams (OMG, 2004: p 2-93). The Actions package is used to define the behaviour of a model via a detailed model of execution (OMG, 2004: p 2-93).

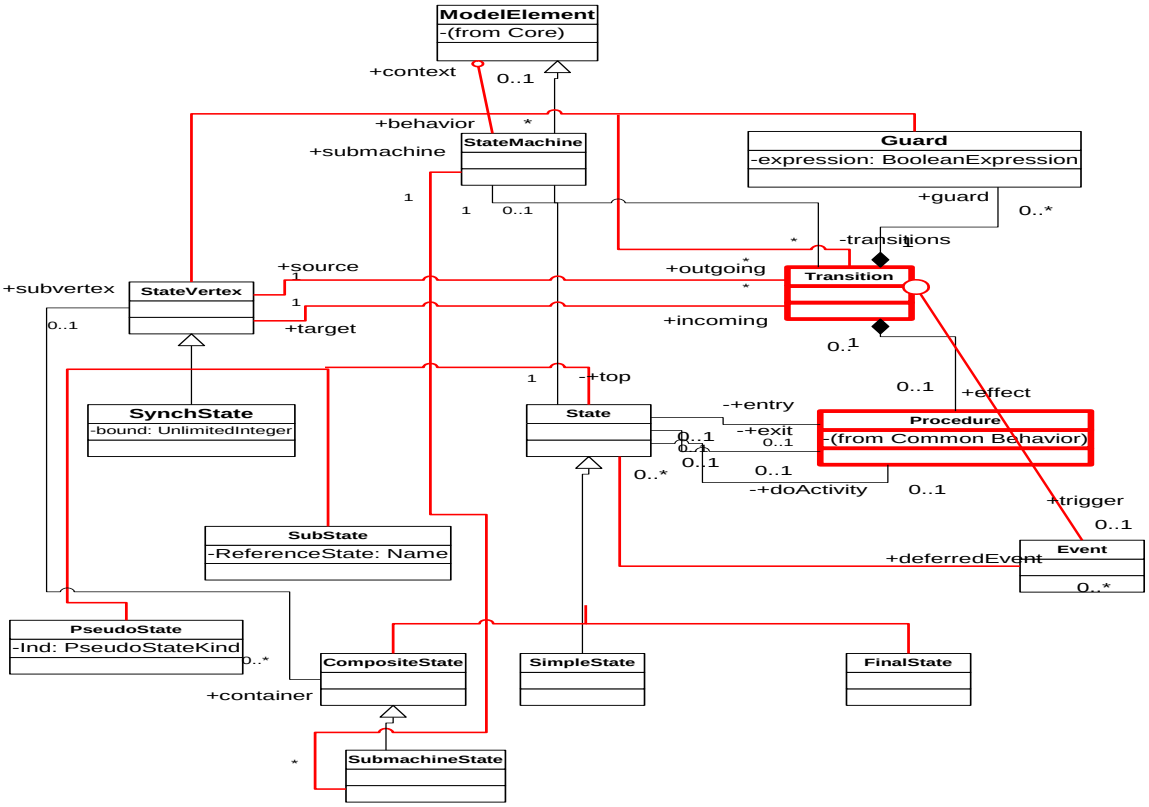


Fig. 3.7.8.3 State Machine Diagram (OMG, 2004: 2-141).

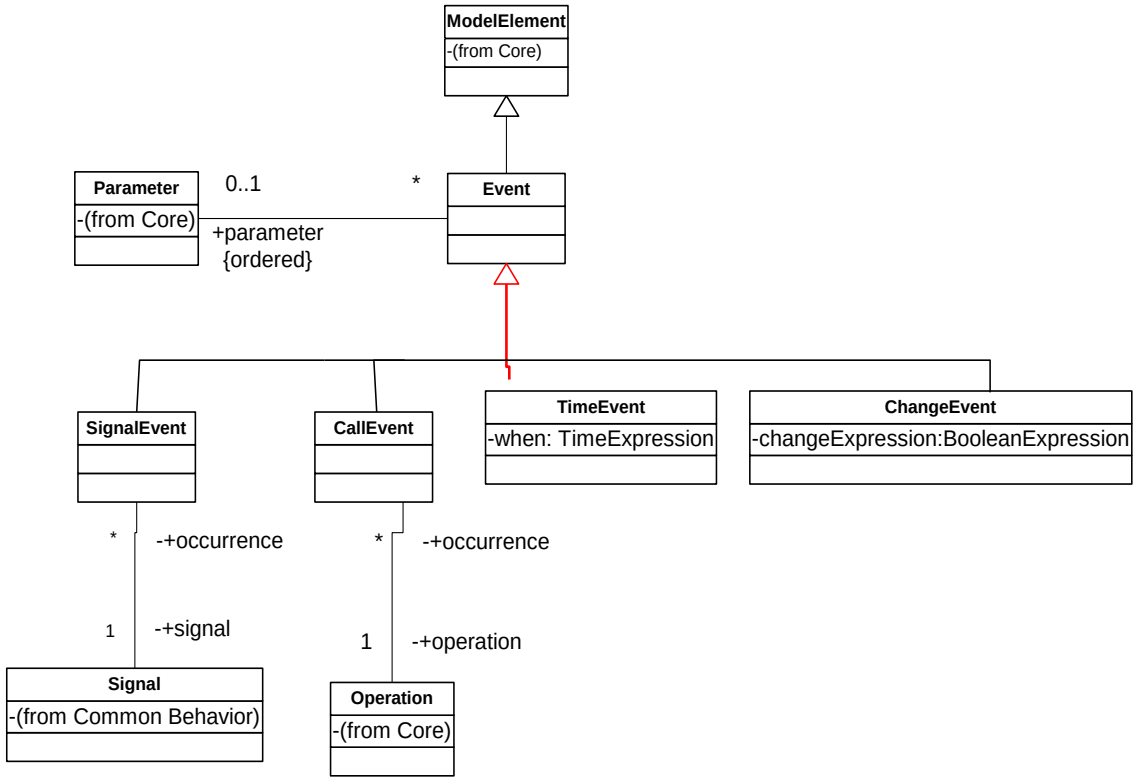


Fig. 3.7.8.4 State Machine – Events (OMG, 2004: pp 2-142).

3.7.9 Activity Diagrams

UML activity diagram describes the dynamic aspect of system. It is essentially a flowchart, showing flow of control from activity to activity. An activity diagram shows the flow from activity to activity. An activity is an ongoing non-atomic execution within a state machine. Activities ultimately result in some action, which is made up of executable atomic computations that result in a change in state of the system or the return of a value. Actions encompass calling another operation, sending a signal, creating or destroying an object, or some pure computation. It models the sequential and possibly concurrent steps in a computational process. It also models the flow of an object as it moves from state to state at different points in the flow of control. Activity diagrams may stand alone to visualise, specify, construct, and document the dynamics of a society of objects, or they may be used to model the flow of control of an operation (Booch, 1999; Rumbaugh, 1999; Larman, 1998).

Activity diagrams describe the internal behaviour of a class method as a sequence of steps. These sequence of steps model the dynamic, or behavioural, view of a system in contrast to class diagrams, which model the static, or structural, view of the system (Muller, 2000a).

An activity in UML represents a step in the execution of a business process. Activities are linked by connections, called transitions, which connect an activity to its next activity. The transitions between activities may be guarded by mutually exclusive Boolean conditions. These conditions determine which control flow(s) and activities are selected (Alhir, 1998).

Activity diagrams may contain action states. Action states are states that model atomic actions or operations. Activity diagrams may also contain events (Alhir, 1998). An ActionState represents the execution of an atomic action, usually the invocation of an operation. Since individual WSL codelines represent atomic operations, these operations are best modelled as ActionStates. Furthermore, an action state is a simple state with an entry action whose only exit transition is triggered by the implicit event of completing the operation associated with the entry action. As soon as the entry operation is complete, the outgoing transition of this action state is triggered. The ActionState is best suited not only to model individual WSL codelines but because it models internal operations that are not triggered by events, ActionStates are well-suited to model batch-oriented systems (OMG, 2004: p 2-171, 2-172).

UML has a degree of specialisation for ActionStates, as well. An example, CallState is a sub-classifier of ActionState. ActionState is an action state that calls a single operation. This specialised modelling element is important in object flow modelling to reduce notational ambiguity over which action is taking input or providing output. By describing the action in terms of a limited set of operations, it is hoped that this description reduces some of the modelling ambiguity within UML (OMG, 2004: p 2-173).

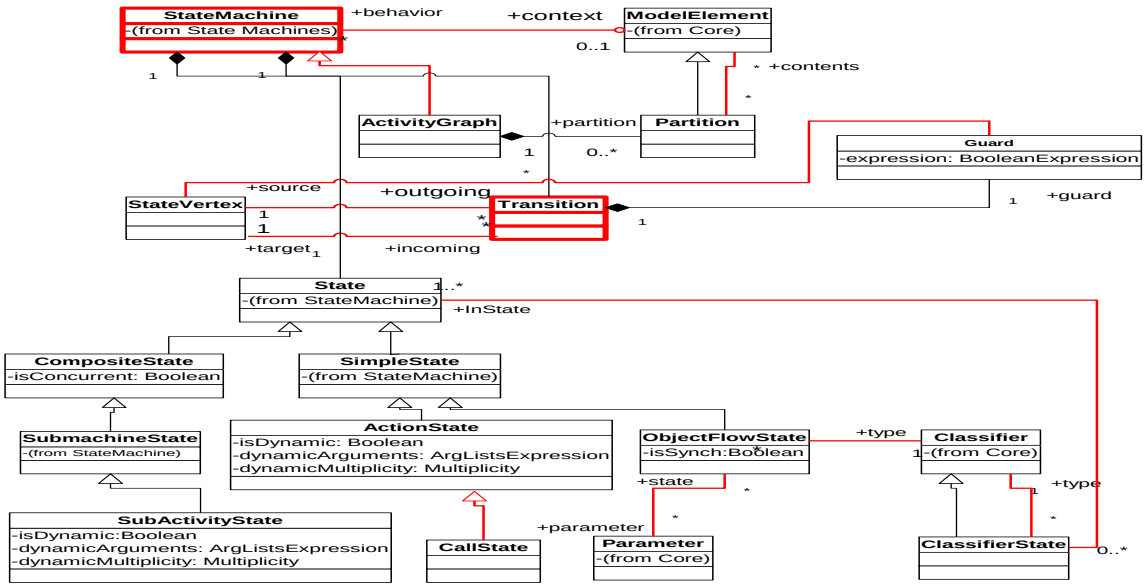


Fig. 3.7.9.1 Meta-Model of Activity Diagrams (OMG, 2004: p 2-171).

Figure 3.7.9.1 has been modified, from the original UML 1.5 Specifications Activity diagram, to include different attached components of the StateChart's StateMachine component such as the Guard and Transition elements. Guard and Transition elements are used in activity diagrams to govern the activity diagram's control flow and to manage parallel and sequential flows respectively.

The Transition structure of the UML Activity diagram is inherited from the State Machines model specification. In the UML Activity diagram, transitions are often implicit (the previous action state, upon completion of its specified action execution, automatically enters the transition to the next action state) (OMG, 2004). However, these transitions may not be simple transitions from one action state to another but may contain guard conditions or be transitions with multiple incoming action states but one outgoing action state (a merge) or the opposite, a join, where a single action state has multiple outgoing action states. Hence, there is a need to explicitly specify transitions within the generated activity diagrams.

Activity diagrams can be partitioned into object swimlanes, or partitions, that determine where an activity is placed in the swimlane, or partition, of the object where the activity occurs (Muller, 2000a).

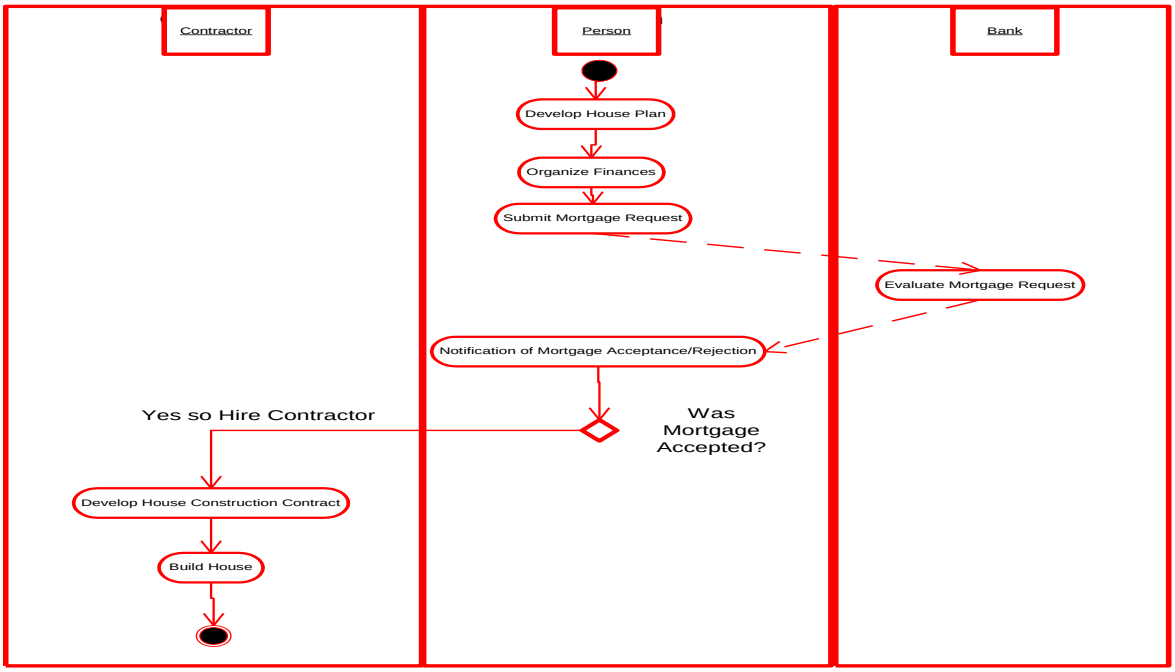


Fig. 3.7.9.2 Activity Diagram of a Person Arranging Financing and Contracting for a House.

The Activity Diagram models the processes, their interactions, and the objects of Person, Bank, and Contractor as a Person plans, applies for a mortgage, and builds a house.

Component diagrams model the packaging of an object as a solution. Deployment diagrams show the physical configurations of software and hardware (Muller, 2000a: pp 122-123).

3.7.10 Advantages of UML

UML has many advantages for a graphical modelling notation. UML encompasses many earlier modelling notations that are well understood and well accepted by the software development community. An example, UML uses statecharts, a modification of traditional state transition diagrams, to model the system behaviour in terms of states, transitions, and events. Consequently, unlike many other graphical notations, the learning curve for UML's notations for its users, who may be familiar with earlier notations like state transition diagrams, may be much less steep (Muller, 2000a).

UML has a core set of concepts that remain unchanged but UML provides a mechanism to extend UML to new concepts and notations beyond this core set. UML allows concepts, notations, and constraints to be specialised for a particular domain (Muller, 2000a).

UML is implementation and development process independent (D'Souza, 1999). In other words, UML is not dependent on any particular programming language nor is it dependent on a particular development process, such as the waterfall cycle software development model.

UML addresses recurring architectural complexity problems using component technology, visual programming, patterns, and frameworks. These problems include the physical distribution, concurrency and concurrent systems, replication, security, load balancing, and fault tolerance (Muller, 2000a).

Besides allowing information to be exchanged between different tools, UML can be viewed as the basis for a new standard method because most of the currently existing methods have been integrated within UML. UML replaces these methods. However, because UML has integrated many different methods in it, it probably encompasses a certain number of redundant or similar concepts. Similarly, because UML is a political compromise between the diverging interests of persons and companies, a number of concepts were included that were not needed but only included for political reasons (Trauter, 1997).

3.7.11 UML Model Exchange Formats

In order to enable the developers to achieve the same understanding and interpretation of the model when the same model is exchanged among different technologies and tools, some sort of common model exchange format is needed. XML is often used for the canonical mapping of the UML model to the data interchange format due to XML's hierarchical structure and linking abilities that enable it to represent a wide variety of information structures. XMI is an XML standard for exchanging UML models and is used by such tools as Rational Rose. Because XMI is very cumbersome in that it often requires a large XML document to represent a small UML model, other standards have been proposed such as UXF. UXF (UML eXchange Format) is a XML-based model interchange for UML models developed by Junichi Suzuki and Yoshikazu Yamamoto (Suzuki, 1999; Taentzer, 2001).

Two sets of standards for these schema conversion rules are XMI and UXF. In this thesis, another method is developed to exchange UML model information, WSL-UML notation. By defining various structures and lists in WSL which are directly linked to their corresponding structures in UML Specification 1.5, it is possible to define a notation that is both compatible with the most recent UML Specification and yet based on the formal notation, WSL. Furthermore, this notation gives the UML diagrams generated by the thesis' tool, TAGDUR, a tool independence such that this WSL-UML notation that they are represented in can be quickly converted into an UML-exchange format acceptable by a particular tool. This tool independence is important when there is no universal UML-model exchange format defined for the many UML tools available (Koester, 2005).

The UML Specification 1.5 was selected because it was the most recent released version of UML. UML version 2.0, at the time of writing of this thesis, had not been formally finalised and released to the computing community.

3.8 UML Components Expressed as WSL Constructs

One of the main problems when using graphical notations, such as UML, with the development of systems is that they are ineffective due to the notation's lack of precise semantics. The lack of precise semantics can cause important properties of a model to be informally verified only, design correctness can not be checked, and a model can be misinterpreted. This lack of precision has been widely recognised and produced a number of approaches (France, 1998).

One approach is to make the notations more precise and amenable to rigorous analysis is to integrate them with a suitable formal specification notation. Most attempts in the approach focus in the generation of formal specifications from less formal object oriented models. These attempts (Bruel, 1997; Johnson, 1996) have some advantages; one advantage is that they allow the specifications to be rigorously analysed, in order to uncover problems that would be missed in the less formal object oriented models. One serious disadvantage is that this approach requires an in-depth knowledge of the formal notation and its proof system, which provides a significant barrier to its adoption in commercial use (Evans, 1998).

Another approach is to extend formal notations with object oriented features. This approach make formal notations more compatible with object oriented notations. Some examples are Z++ and Object-Z (Lano, 1994; Duke, 1991). One disadvantage of this approach is that although a number of formal systems exist, they have the disadvantages of being too different from current industrial methods to be suitable for general industrial application and of lacking available analysis tools (Evans, 1998).

Liu extended a formal notation, WSL, to adopt object-oriented concepts (Liu, 1999). The formal notation of WSL has been used for general industrial applications for a number of years (Ward, 2001). Furthermore, by encompassing this notation within various tools, such as Maintainer's Assistant/Fermat (Ward, 1989), and TAGDUR (Millham, 2003b; Millham, 2004; Pu, 2003; Pu, 2005), the disadvantage of a lack of available analysis tools is reduced.

3.9 Summary

Several topics relevant to this thesis are introduced in this chapter. Formal methods are introduced as a way for software assurance during the reverse engineering process. A formal notation, WSL, was introduced and reasons for WSL's adoption over other formal notation are given. The need for program understanding, and consequently software visualisation, is explained in order to enable developers, among other system stakeholders, to more fully understand the system to adapt it to new requirements. The topic of domain engineering and knowledge was introduced in order to both explain its role in program understanding and the need to abstract program specifications from the source code. UML diagrams were selected, for several reasons provided, to visualise a program's structure, functions, dynamics, and architecture. A description of the various UML diagrams, providing different views of the system, is given.

Chapter 4: Proposed Approach

4.1 Introduction

The purpose of this chapter is to multi-faceted. It first explains the background of the selected legacy system of this thesis and its needs, from a business standpoint, for reengineering. A brief summary of the restructuring and UML diagram extraction processes that are used in this thesis are provided with more details of these processes, including algorithms, provided in future chapters. WSL is extended, through the WSL-UML notation as described in this chapter, to represent the abstract modelling notational elements of UML. This WSL-UML notation is exported, via the XML exchange format of XMI, to visual UML model diagrams through the use of tools. The purpose of these visual UML diagrams is to serve as a means, via a UML checklist (see Appendix H) to ensure completeness and representational accuracy, of the validation of extracted UML diagrams from the source code using this thesis tool. Once this UML diagram extraction process is validated, these UML diagrams may serve other purposes such as increased program understanding for the developer and the extraction of business rules.

4.2 Definition of Batch Oriented Systems

The selected legacy system is a COBOL, mainframe-based legacy system that operates in a batch-oriented fashion.

A batch-oriented system has usually the following set of characteristics:

- Limited, if any, user interface
- Limited and clearly defined set of inputs/outputs
- Inputs and output arrive and leave in a predictable fashion
- A clearly-defined sequence of steps to be followed during processing
- Do not receive nor handle events
- Usually do not possess timing constraints on their processing

In contrast, a highly-reactive system will tend to respond to a wide range of possible events, which can arrive in an unpredictable fashion, and handle a diverse set of inputs rather than respond to the arrival of a clearly-defined set of inputs. Consequently, a highly reactive system's processing of these events, and consequent output/responses, is highly variable and dependent on its input and events (Wang, 1993).

4.3 Background of Selected Sample Legacy System

The original legacy system from a telecommunications company is a batch-oriented system. This system was designed to model an old business process. Under the old business process, a clerk would receive a request, for a customer service order, via mail. The clerk would formulate a request record and put it into a batch-input queue for processing. This record would be processed by a batch process during the following night. The resulting output record would be sent back to the clerk who would, in turn, inform the customer that their customer order would be filled (Telecommunications Employee B, 2002).

The telecommunications company originally built separate systems for each part of its business. It had separate sets of systems for each business section such as customer relationship management, billing, maintenance, etc. Systems were not integrated and these systems did not always communicate with one another. As a consequence, many problems resulted from this lack of integration. An example, a customer desired a certain type of telecommunication service so they called a customer care clerk to arrange for a customer service order. The clerk entered the request into her customer care system which sent a request to the customer billing, which billed the customer for the call, and supposedly passed on the request to the dispatch system, which dispatched a telecommunications worker to the customer to handle this customer service order request. However, these systems were poorly integrated and requests were often lost or not sent at all. Consequently, often a customer booked an appointment with customer care and waited at home to receive the telecommunication employee at the scheduled time. In the meantime, the request was not sent to the dispatch system from the customer care system so a telecommunication's employee was not dispatched. The customer then waited in vain and then later, the same customer is billed for this non-existent service call. Because of this non-integration, mistakes were often made and customer dissatisfaction resulted (Telecommunications Employee A, 2002).

When the Telecommunications Company automated its business processes during the 1960s and 1970s, the Telecommunications Company developed independent systems for each set of business processes. Customer care business processes were handled by one independent set of systems; customer-billing processes were handled by another independent set of systems; and service employee dispatch processes were handled by still another independent set of systems. These systems were not designed as part of a comprehensive information systems plan with a view to providing an integrated enterprise-level business information system; instead, these systems were developed, for the most part, independently and on an ad-hoc basis in order to address a specific business need. Only after these systems were operational was the need for integration of these systems recognised and then efforts, often ineffective, made to allow inter-communication among the existing systems (Telecommunications Employee A, 2002).

Many of these systems were batch process systems that modelled earlier business processes. These early business processes were modified to accommodate the computational resource limitations of that time period. During the 1960s and 1970s, if a telecommunication employee had a system request, such as create a new customer account, this system request could not be executed immediately but had to be queued in a request list (COBOL input cards) for processing in batch later on. These legacy systems retained their batch system nature even though the business nature and computer resources changed to allow online, real-time processing. Because these legacy systems fulfilled a critical business need and because of the high development and switchover cost associated with new system development, these legacy systems were retained. The original COBOL interface was modified; screen scraper applications obtain user input and send it, as batch input to these legacy systems; screen scraper applications then retrieve the results from this batch job from the legacy system and then display the results to the user (Telecommunications Employee D, 2002).

Another factor was that the telecommunication company had centralised its data processing with these legacy systems. Data processing was handled by two central locations in Calgary and Edmonton. However, with recent mergers and acquisitions and the decentralisation of the telecommunication company offices, data access and processing must be handled from several different and geographically disparate locations. To address this need, there is a need to reengineer these legacy systems into a

client-server model, with clients at each location, or to Web-based multi-tiered architecture model. Both models require a system transformation (Telecommunications Employee B, 2002).

As business needs changed, there was a need to replace the old batch model with real time processing model. This need for replacement is especially true if these systems move to a Web-based model. Many businesses, such as certain cable companies, are moving to a Web-based interface for their customer care service in order to enable customers to request services themselves and by having self-service capability implemented, this self-service capability reduce the company's employee costs. One requirement for this Web-based interface is real-time response times. If a customer, using this Web interface, requests a service or inquires about their account, they can not be expected to wait several hours, which is the typical response time of this legacy system, for the system to process their input and return the results of this processing to these customers via this Web interface. Web-enabled architecture typically demands an object-oriented, event-driven architecture which is incompatible with the procedural driven and structured nature of this legacy system (Telecommunications Employee D, 2002). Wrappers could enable this legacy system to interface well with a Web-based front-end, and at the same time, reuse the trusted functionality of legacy systems and retain their massive investment in the legacy system. However, enclosing legacy system in wrappers is often viewed as a short term solution because wrapping does not address many of the problems that legacy systems often entail, such as overloading, static functionality, and high maintenance costs (Bisbal, 1999).

The original legacy system, consisting of approximately 112 000 lines of COBOL source code, was designed to run on an IBM 370 mainframe computer. The original COBOL dialect that this program was written in was a specific COBOL dialect that was compatible with this specific IBM platform (Telecommunications Employee C, 2002). There are many different COBOL dialects; often a particular COBOL dialect was designed to run on a particular set of machines. If one wanted to port COBOL program to another platform and this platform did not have a COBOL compiler that could compile the COBOL program's original COBOL dialect, the program would have to be rewritten in a COBOL dialect that was compatible with the new targeted platform. If this COBOL program was translated into standard C++, this portability issue would not arise. C++ compilers are available for many platforms, from mainframes to UNIX boxes to Intel-based machines, so that program written in standard C++ could be run without modification on any platform supporting a C++ compiler. The issue of having a program tied to a specific set of platforms, such as the IBM 370 set of machines with the ensuing costs of being tied to a proprietary piece of technology such as the purchase of proprietary hardware and proprietary operating system software maintenance contracts, is not an issue if this program is translated into C++ with its platform-independence advantage (Siff, 1996).

Another problem arises in that this COBOL program must be maintained by programmers fluent in the particular dialect of COBOL that this program was written in. COBOL programmers are a dying breed; having a corporation depend, for the maintenance of their business-critical information systems, on an ever-shrinking pool of COBOL programmers puts them at a distinct disadvantage. On the other hand, C++ is much more standardised with far fewer dialects and with much less differences among dialects than COBOL. C++ is the most popular programming language in the world; the pool of C++ programmers is quite large (Rayson, 1999). Although translation to C++ from a WSL representation has not been validated, a nascent, and unvalidated, ability to translate WSL to C++ using this thesis tool is available (see Appendix D for WSL to C++ rules).

Data manipulated by the legacy system is native to that system; the legacy system was designed to define and manipulate this data in a standalone mode without having other systems access this data. This standalone data model is compatible with the original

independent development of these systems; however, this data model is incompatible with the current, inter-dependent nature of enterprise systems of a corporation. An example, the customer billing system contains its own data regarding the amount of each product used by each customer. Although the customer care system was not designed with this data mining purpose in mind, this data is highly useful for data mining by marketing and management systems which use this data for determining and tracking market trends (Telecommunications Employee A, 2002).

The major feature of the reengineering effort is representing this system in UML. For many legacy systems, the source code is the only system documentation that has survived. Changes made during the long maintenance history of a program have obfuscated the original design. At the same time, the need to integrate these formerly standalone systems creates a great need to completely understand the complete legacy system, not only its internal structure and behaviour but also its interaction with file systems and with other external entities. This tool parses the source code to obtain a UML representation of this system; this UML representation embraces both the dynamic and static views of the system through UML activity, class, statechart, sequence, and collaboration diagrams. Other reengineering tools extract the static view only from a legacy system in the form of class skeletons and procedure headers (Rational, 2002). There are two major problems with these class diagrams. One problem is that most legacy systems, especially those written in COBOL, are procedural rather than object oriented and that most reengineering tools do not have the capability to transform a procedural system into an object-oriented system. The other problem is that the static view does not help the developer to fully understand the system being reengineered; the static view provides a list of procedure headers but gives no information on how these procedures interact with each other or how these procedures interact with external actors such as databases. Furthermore, the static view does not model how data flows through the system nor does it represent the control logic of the system. The procedure headers themselves are meaningless unless the developer has maintenance experience of the system being reengineered (Kwiatkowski, 1999).

A literature survey and search of existing tools revealed that few academic or commercial reengineering tools restructure existing systems into a new architectural model such as object orientation. No reengineering tool provides a full re-documentation of a COBOL system, through a series of UML diagrams, along with restructuring to an object-oriented, event-driven system (Embercardo, 2004; Renaissance, 1999;Muller, 2000b).

4.4 Thesis Approach

In this thesis, a unified approach for redocumentation is proposed. The approach is based on the use of the wide spectrum language, WSL, which has the advantage of a sound formal semantics and basis. The scope of research includes:

- a set of rules or methods for translating COBOL legacy systems into WSL
- a set of transformational processes to transform a procedurally-structured, procedurally-driven system in WSL to an object-oriented, event-driven system
- restructuring the procedural system into an object oriented one. This restructuring involves several steps:
    - the first process is to identify the optimal clusters of data and the operations that manipulate them and transform these clusters into classes

- The next step is to identify which tasks, defined within these operations, can operate independently and which operations must execute sequentially. This information is then used in the next step to determine which events are synchronous and which events are asynchronous.
- The third step is the process of identifying pseudo-events from the legacy system and then determining whether each event is synchronous or asynchronous.

- UML diagrams are then extracted from the source code using, in part, the information gained during this restructuring process. For example, asynchronous events, which are identified during the event identification process, may be modelled as events among classes.

New constructs in WSL are formulated, to represent constructs within COBOL, in order to represent an object oriented, event-driven system and to represent the UML diagrams via their modeling elements in a WSL-UML notation.

A set of rules is derived to represent UML diagrams from this transformed system represented in WSL. These UML diagrams include:

1) class diagrams
2) activity diagrams
3) sequence diagrams
4) deployment diagrams
5) component and deployment diagrams

The steps of these restructuring and UML extraction processes are briefly outlined in the chapter along with various methods of others and why their approaches were unsatisfactory for our reengineering purposes.

4.5 Conversion from COBOL to WSL

The thesis' approach consists of reverse engineering a batch-oriented COBOL legacy system into the Wide Spectrum Language (WSL). The rules for COBOL to WSL conversion were formulated using Martin Ward's (Ward92) paper, *The Syntax and Semantics of the Wide Spectrum Language,* which defined the basis of the Wide Spectrum Language. Due to the huge number of possible COBOL constructs and the large number of different COBOL dialects, this conversion from COBOL to WSL had to be done manually. More explanation of the particular problems of this conversion process and why a manual conversion process was chosen is given in chapter 5.

4.6 Restructuring Procedural WSL

4.6.1 Implications

4.6.1.1 Implications in UML Modelling

This restructuring from a procedural to an object-oriented system is important for two reasons. One reason is that it enables the legacy system, represented in WSL, to be moved to a platform usually requiring an object-oriented architecture, such as a Web-enabled platform. Another reason is that the information gained from this restructuring phase is used in the modelling of the restructured system. An example, the clusters of procedures and variables, which were identified during the object identification phase, become class diagrams with methods and attributes respectively (Muller, 2000a).

4.6.1.2 Implications in Business

Restructuring a procedurally structured system into an object-oriented system has many business advantages. By compartmentalising related data and procedures together into modules called classes, developers are able to understand the system as a series of distinct but related pieces. Pieces, or classes, along with their relationships, can be understood by the developer much more easily than a huge, monolithic system. Furthermore, object orientation has a distinct advantage in certain architectures such as Web-based platforms where the system, through its class collection, may be distributed among several tiers or machines. Another advantage is component reuse. A class or object of a legacy system may be taken out, modified, tested, and then replaced with little overhead and downtime. Furthermore, this object or component may be reused by different applications. In the monolithic procedurally structured approach, the entire system must be taken out (or a copy of it thereof), modified, tested, and then the entire system, not just a small component, must be replaced. This replacement takes considerably more downtime and overhead than a component replacement. Furthermore, while an object component may be reused by several different applications, a procedure within a procedurally structured system can only be reused within the same application in which it is defined (Flint, 1997).

Gall identifies the reasons for restructuring of legacy systems as a means to control spiralling maintenance costs, improve system performance, and transition the system to a distributed and more efficient hardware environment. In order to adapt to constantly changing business needs, the business must have to look at alternatives, such as reengineering, rather than replacing or developing systems (Gall, 1995). With restructuring to an object-oriented structure, the technical and economical lifetime of a system is extended and the system's software ageing cycle can be better controlled (Parnas, 1994).

One argument against restructuring as a reason to move the original procedurally structured, mainframe-based platform on to a new platform, such as a Web-based platform, is that the original system could be more easily enclosed in an object wrapper. However, in order to enclose a system in an object wrapper, it is necessary to understand the structure, behaviour, and dynamics of the system in order to develop a proper interface between the original legacy system and the object wrapper (Bisbal, 1999). These gaps in knowledge, which are needed for object wrapping, can be addressed by the thesis tool, TAGDUR, as it extracts this information from the legacy system source code and then models it through a series of UML diagrams.

4.6.2 Object Identification
4.6.2.1 Method of Object Clustering

In order to obtain the original high-level structures of a program from its legacy source code, object identification methods were developed based on the models of those of Gall, van Deursen, and Newcomb (Gall, 1998; Newcomb, 1998; van Deursen, 1999a). Gall identifies potential objects from source code by measuring the degree of coupling between different entities such as procedures and variables. If two procedures have a high degree of coupling, or interaction among themselves, these two procedures probably should be assigned to the same object class. Gall also uses data dependencies between procedures and variables in order to identify potential objects. If two variables share a common data dependency, such as both of them belonging to the same record structure, these two variables should probably be assigned to the same object class (Gall1998). Newcomb uses the degree of variable coupling, at the COBOL record level, for object clustering. Records, with their fields, with a high degree of coupling among themselves are grouped in the same object class (Newcomb, 1998).

Using one of van Deursen's object identification techniques, it was determined which procedures have a high fan-out (procedures which call many other procedures) and which procedures have a high fan-in (procedures that are called by many other procedures). Procedures with a high fan-out are usually control modules and thus should be kept in a separate control class. Similarly, procedures with a high fan-in, such as error-logging functions, should likewise be put in their own separate classes (van Deursen, 1999a).

Although combining these error-logging functions within their most often-used class object may reduce the number of inter-object couplings, it has the undesirable effect of increasing coupling between the class containing these error-logging functions and other classes whose only commonality with the former class is through the logging functions. Hence, an obscuration of the real purpose of the function within the system design is created when the high fan-in and fan-out procedures are included in the data clustering analysis (van Deursen, 1999a).

Another part of this thesis' object clustering process is the separation of external procedures into their own class. External procedures are procedures, which may be called in one program file, but are defined elsewhere, either in another program file or within another system itself. By separating external procedures into their own classes, a separation is made between procedures that are defined within a program file and belong to this file by some sort of programmer-defined logic and external procedures, which should belong to the system or file where they were defined.

By using a similarity/dissimilarity matrix, variables that are used within WSL procedures will be assigned to objects whose attributes or variables they access the most frequently within the procedure. If a variable is assigned to one class object and a procedure, assigned to a different class object, accesses that variable, an object class association is created between the two object classes. If a procedure/variable of one instance of one class access the procedure/variables of multiple instances of another class, this access is modelled as a one-to-many association. If procedures/variables of many instances of one class access the procedure/variables of many instances of another class, this association is depicted as a many-to-many association. However, since the regineering effort of this thesis produces one object per class, all association are modelled as 1 to 1.

The thesis approach in object identification is use individual record fields, not records, as entities of evaluation for classes, unlike Newcomb's and van Deursen's methods. Gall's degree of coupling algorithm is used to determine which variables and procedures are most closely coupled to specific classes. Van Deursen's algorithm that places controller and logging functions into separate classes eliminates classes whose attributes/methods contain only functional cohesion with other classes. By identifying objects and any associations, whether procedural calls or shared variables, UML class diagrams, with inter-class associations, can be derived. The thesis' object clustering technique was based on a combination of methods of Gall, Newcomb, and van Deursen, along with the separation off externally defined procedures into separate classes (Gall, 1998; Newcomb, 1998; van Deursen, 1999a). More detail regarding this particular technique is given in Chapter 6.

4.6.3 Independent Task Evaluation

4.6.3.1 Development of Independent Task Evaluation Method and Algorithms

The independent task evaluation and method algorithms involve evaluating the selected granular unit and its immediately successive granular unit in order to identify control and data dependencies between them. If any dependencies are determined, then the selected unit is deemed to require sequential execution; otherwise, if no dependencies exist, the select unit may be executed in parallel with its successive unit. More details regarding the particular algorithms for specific granular units may be found in Chapters 7 and 8.

One particular advantage of using WSL as an intermediate language representation in our reengineering effort is that a granular unit, the codeline, can be unified. When evaluating task independence using the task granularity of individual codelines, the concept of a codeline differs greatly among programming languages. A high-level programming language's codeline can perform the same task as multiple lines of assembly code. Thus, the degree of task independence using the granularity of individual lines of code in a high-level programming language like COBOL may not be necessarily similar to the task independence using the granularity of individual lines of code in a low level programming language like assembly language. However, if the source code has been converted to WSL and the evaluation of task independence is performed on this WSL code, an individual codeline remains the same regardless of whether the source code was COBOL, assembly, or FORTRAN.

4.6.4 Event Identification

4.6.4.1 Development of the Event Identification Process

The process of event identification, the last of the thesis' restructuring process, is to identify pseudo-events from the legacy source code and to determine the degree of asynchronicity of each of these pseudo-events. Pseudo-events may be procedure calls, I/O operations, error invocations, or system interrupts. These pseudo-events are then represented in a sequence diagram as message passing between objects. An example, if a method of object A calls another method of object B, this method/procedure call is represented as a call event; this call event is represented as a call message between objects A and B in a sequence diagram. To model a message in a sequence diagram, this message must be represented as asynchronous or synchronous. In order to determine this message's asynchronicity, it is necessary to determine whether the task incorporating this method call and its immediately succeeding tasks can execute independently or whether they must execute sequentially. If the task incorporating the

method call can not execute independently from its immediately successive task, this message is modelled as a synchronous message; otherwise, this message is modelled as an asynchronous message.

The independent task evaluation and event identification processes are important in the modelling of various behavioural diagrams, such as UML activity diagrams. An UML activity diagram represents the dynamic view of a system as a sequence of steps grouped sequentially as parallel control branches. An activity diagram is similar to a statechart except that a statechart distinguishes between states, activities, and events while an activity chart focuses on activities. In order to model parallel activities within an UML activity chart, it is necessary to first determine which tasks, which will be modelled as activities, may be executed in parallel and which tasks must be executed sequentially.

Non-independent tasks will be modelled as sequential activities and independent tasks will be modelled as parallel activities. The definition of parallel processes is a process that can be executed in any order. Synchronisation bars are used to synchronise the divergence of sequential activities into parallel tasks or the merging of parallel tasks into a sequential task. These synchronisation bars enable the control flow to transition to several parallel activities simultaneously and to ensure that all parallel tasks complete before proceeding to execute the next sequential task.

Events, identified during the specified event identification process, are also modelled in the activity diagram. Events are modelled in activity diagrams. They are depicted as having their origin in the activity where they occur and are depicted as having their destination in the activity in which the event is invoked. Depending on whether the specified independent task evaluation algorithm, using individual code lines as their granular unit as per the defined activity unit, deems these events synchronous or asynchronous, the events are modelled as such.

4.7 OCL

4.7.1 Introduction

In this section OCL, which is used to express specifications of a UML model, is introduced. OCL is not strictly a formal notation but is more of a rigorous notation. Although OCL has several advantages, it was deemed to be unsuitable for our thesis' modeling purposes for several reasons.

A constraint can be defined as a rule that allows you to specify some limits on model elements. Often, these rules are specified in a natural language, which is prone to ambiguity. Consequently, UML provides a language, the Object Constraint Language, which is provides the rigor that is missing in natural language specifications yet is rather simple to construct or interpret (Bennet, 2001: 250-251).

In UML diagrams, constraints are expressed as a text string, enclosed by braces, which is placed near the element or connected to that element via a dependency relationship. An example of a constraint, context CarDriver inv: age>=14. The key word context introduces the model element, CarDriver, to which this constraint applies and inv introduces this constraint as an invariant. Given this constraint, it means for the UML component, CarDriver, all drivers must be over 14 years of age (Bennet, 2001: p 251). The context of an invariant must be a class for invariants or an operation (for pre and post conditions). Furthermore, the class,

providing the context for the constraint, may be nested within one or more packages. The path that specifies this packaging uses the following format: EnclosingPackageName::TypeNameOrClassName (Bennet, 2001: p 254).

4.7.2 Features of OCL

OCL has the following features:

1. Invariants – properties which must remain true throughout the life of a model element, such as an object. Invariants specify the constraints under which a system can operate

2. Pre-conditions – a pre-condition is a condition that must be true before a particular part of the system is executed. The purpose of pre-conditions is to perform checks before an operation is executed and to prevent the system from entering illegal states

3. Post-conditions – a condition that must be true after a particular part of a system is executed, assuming that all of its pre-conditions were met so that it entered a legal state and that the system successfully carried out its action. Post-conditions are used in conjunction with pre-conditions to formulate rigorous proofs that a particular piece of code successfully executes (fulfills its post-conditions) provided that any of its pre-conditions, if present, are met.

OCL also allows the nesting of model elements.

4.7.3 Advantages of OCL

OCL is designed to provide a clear and unambiguous method of describing rules about the behaviour of elements in a set of UML models. Constraints are used to specify pre-conditions and post-conditions on use cases and operations, to describe invariants in operations, to describe guards on transitions, and to describe invariants for classes and for types, in the class model. OCL was designed to provide a clear and unambiguous language for the description of constraints and to specify accurately the behaviour of elements of the model (Bennett, 2001: p 253).

OCL is an expression language with does not change the value of any element of a model. One can think of OCL as conditions that determine control, such as condition statements in an if statement, rather than as instructions to perform an operation (Bennett, 2001: p 253).

In several ways, OCL is ideal for use in the expression of UML model notations via WSL. The use of invariants as guards is very similar to the use of guards to control the flow of action in both activity diagrams and statecharts. The pre and post conditions as expressed in OCL are very useful in expressing the pre-conditions that a transition must satisfy before entering a state in a UML statechart and in expressing the post-conditions that are met when a state in a UML statechart is reached. However, the inclusion of modelling the sample system via UML statecharts is outside the scope of this thesis.

4.7.4 Unsuitability of OCL for Thesis' Modeling Purposes

Unfortunately, OCL is a typed language and, subsequently, OCL statements must conform to typing rules such as not comparing strings to integers (Bennet, 2001: p 253). Elements of the OCL expressions themselves are expressed in terms of types. The four

basic types of OCL are Boolean, Integer, Real, and String. These OCL types may be combined using operations peculiar to their type. An example, the Boolean type may have the following operations of and, or, or implies while the Integer type may have the operations of multiply, add, or subtract. This typed nature of OCL makes it unsuitable for WSL. WSL, by its nature, is typeless. Expressing constraints in a WSL version of OCL would entail the imposition of types on WSL and its consequent constraints on the type of operations, such as concatenation or multiplication, that these typed variables in WSL would be allowed to perform. Introducing types in WSL would reduce its advantage of platform independence in that data typing is often very platform dependent. An example, the way that C programming language implements a variable of type string is quite different from that of Visual Basic.

4.8 Source Code to UML Reengineering

Trauter's reengineering approach includes modelling elements from UML that are needed for reengineering as well as the addition of elements that are needed for reengineering purposes but are not present within UML. This approach has two options. One option is an academic approach where a theoretical analysis is performed on UML in combination with a more precise description. The other option, favoured by Trauter, is an industrial approach where the UML meta-model is taken, implemented for the reengineering repository, and improved iteratively (Trauter, 1997).

Demeyer defines round-trip engineering as a seamless integration between design diagrams and source code; a programmer generates code from a design diagram, changes that code in a separate development environment, and then regenerates the new design diagrams from the modified code. Round-trip engineering involves a succession of processes where each process involves some reverse and forward engineering aspects. Because round-trip engineering demands for a tight interaction between reverse and forward engineering, the supporting tools need an adequate representation of source code. These tools need to incorporate information regarding classes, methods, attributes, and associations. These associations include both static belongs-to classes and dynamic invocation relations between methods. UML presently does not incorporate two concepts, the method invocation and access attribute, which are necessary constituents of the implementation model. Demeyer suggests extensions of the UML model in order to represent these two concepts, such as a stereotype of an association element to represent method invocation (Demeyer, 1999).

Although there are existing tools that reengineer legacy systems using WSL as an intermediate representation, the thesis' tool, TAGDUR, has many additional advantages than Fermat. TAGDUR provides an ability, not fully developed, for round trip engineering by translating the transformed and re-documented COBOL programs to C++, a more modern language (see Appendix D for WSL to C++ conversion rules). TAGDUR extracts the specification of components of the system. The intermediate representation of this component in WSL is available for later transformations, re-documentation, and recoding. The original specifications of the system may be extracted during reverse engineering; the original program may then be re-engineered for another architecture (object-oriented) and platform (C++). TAGDUR differs from FERMAT in that it offers several additional features. TAGDUR has the ability to transform the originally procedurally structured program into an object-oriented system.

In this section, descriptions of methods in this thesis to extract various UML diagrams are provided. Further details, such as extraction algorithms, are provided in later chapters.

4.9 Extraction of UML Diagrams

4.9.1 Introduction

In this section, methods to extract various UML diagrams are described with further details, including the appropriate algorithms, outlined in later chapters.

4.9.2 Method to Extract Class Diagrams

Using the object identification algorithm clusters of variables and procedures are identified and put into class objects. Procedures from one class object that access a variable/procedure of another class object form an association between class objects. These associations are available from the variable/procedure usage matrix used in the object identification algorithm outlined in this thesis. These matrices determine the navigability of the association. An example, if a procedure P of class object A accesses the variable/attribute of class object B but class object B does not access any variable/procedure of class object A, then this association is a one-way navigation between object A and B. If class object B did access any variables/procedures of object A, then this association would be a two-way navigation association.

The usefulness of the object identification and association identification among these classes are that it enables developers, particularly in a multiple machine environment, to put closely-coupled classes together in order to reduce network traffic and enables them to determine which class might access another class on another machine in what manner and in what frequency. This remote access often involves special programming and this association identification helps the developer plan for these accesses (Wijegunarante, 1998).

The stereotypes for classes such as define, realise, trace, and refine are not of particular use for reengineering purposes. An example, traces are used to track changes across models (OMG, 2004). Trace stereotypes may be very useful during the software evolution of an object-oriented system but this selected system originally had no classes so the reengineering effort produces original class objects with no history. Consequently, the object-oriented system that this tool produces has no use for trace stereotypes. Consequently, this WSL-UML notation, for simplicity, excludes class stereotypes.

PackageList – list of all packages, by their package name, within a system. It is important to incorporate all source files within a system into a list or structure so that they can be accessed more easily (OMG, 2004).

Package is a container that contains all related model elements of a system, subsystem, or other type of system partition. In class diagrams, the Package would contain the list of classes, list of associations between these classes, and association ends that connect classes with their associations (OMG, 2004). In the specified UML derivation algorithms, the object and association identification algorithms were based on source files, which correspond to COBOL copybooks.

Association defines a semantic relationship between two classifiers, such as Classes. Each Association must have a unique name, as an identifier, within its enclosing package. An Association has at least two association ends, each end associates a classifier with the Association. An instance of an association, which connects instances of classifiers, is a link (OMG, 2004: pp 2-19, 2-22).

An association class is an association that is also a class. It connects both a set of classifiers, like an association, but also defines a set of features that is associated with the relationship itself and not any of the classifiers (OMG, 2004, pp 2-21). Because reengineering identifies rather simple associations, association classes are not typically needed in the reengineering effort and thus, these association classes are not included in the WSL-UML notation.

More details regarding the methods and algorithms to extract class objects are available in Chapter 6.

4.9.3 Method to Extract Activity Diagrams

Because these UML activity diagrams are code-based, the conditions contained within the control constructs form guards for the activities and WSL code operations form activities. These activities, as embodied in an individual WSL codeline, are associated with the ActionState.

Because WSL is platform independent, many platform-related details such as memory allocation, deallocation, and file handling are excluded from WSL and, consequently, the activity diagrams concentrate on how data is manipulated within the program and on the program's control flow. The related WSL codeline is provided for reference purposes. Because WSL is not fully understood by many developers, a description of the activity present within the activity itself was included. An example, the activity's operation is described as a method invocation, assignment, file read, or file write. In the case of a file read or write, the destination, source, and index variables are specified in order to clarify the meaning of the activity.

Additionally, some of the attributes of an ActionState such as dynamicArguments and dynamicMultiplicity can not used within a platform independent context. The ActionState, isDynamic, is a Boolean flag which indicates whether the ActionStates arguments might be executed concurrently. The thesis' independent task evaluation algorithm indicates whether a given ActionState may be executed concurrently or not; if so, this attribute is set to true, otherwise, it is set to false. However, the other ActionState attributes, such as dynamicArguments, determines, at runtime, the number of parallel executions of the actions of the state (OMG, 2004: p 2-172).

The Event structure of the generated UML Activity Diagram, even if it models a pseudo-event such as a method invocation, is inherited from the State Machines model specification (OMG, 2004). It is important to first identify and then model these events within the Activity Diagram in order to transition this system from a purely procedurally-driven architecture to a more event-driven architecture.

The following diagram models the following WSL code sample as action states in an activity diagram. Each action state is labelled by the WSL statement whose entry action the state represents. Each action state is placed in the object swimlane whose object

produces the action. An example, the invocation of Class B's UpdateRec method is represented as an action state in the Object B swimlane. Potential parallel operations, such as "M := N + 2" and "K := 3", are modelled as parallel flows emanating from a fork synchronisation bar. An if-then-else WSL construct is modelled in the activity diagram as a branch, with mutually exclusive guard conditions, proceeding to two action states. Using these guard conditions, the control flow in the activity diagram is governed in a similar manner to the system that it represents. Potentially parallel executing WSL code lines are modelled as parallel control flows in the activity diagram (Millham, 2003b).

More details regarding the methods and algorithms to extract activity diagrams are available in Chapter 8.

4.9.4 Statecharts

Statecharts model the dynamic behaviour of a system; unlike activity diagrams, which model the flow between areas of work, statecharts could be more accurately described as modelling the changes between states of an instance. In the selected legacy system, which is both batch-oriented and stripped of a user interface, outside interactions/events are few. Consequently, an activity diagram, which models work flow based on source code, is of most help to a developer in order to understand the dynamic behaviour of a system (Muller, 2000a).

Statecharts are of most help when developers need to understand the behaviour of highly reactive systems that are subject to a wide range of external stimuli and/or events. Since the selected legacy system possesses neither of these characteristics, it was decided that the system's dynamic behaviour would best be described via activity diagrams, rather than statecharts (Alhir, 1998). Consequently, a method to extract a statechart from source code generally is not feasible in a batch-oriented system.

However, both activity and statecharts share common UML modelling notations such as events and transitions. In the case of transitions, in activity diagrams, the incoming transition list is a list of one or more activities while in statecharts, the outgoing transition list is a list of one or more states. The incoming-outgoing transition list is similar in both statecharts and activity diagrams (Muller, 2000a).

In reengineering, particularly reengineering batch-oriented systems, the type of events that would be of most use would be the Call Event where a procedure or operation is invoked and possibly, a Signal Event when a System Error or Interrupt occurs. ChangeEvent, which models an event that occurs when the specified boolean condition becomes true as a result of a change in value of one or more attributes, is designed to model more of an event-oriented, reactive system (OMG, 2002: pp 2-142-2-143).

The type of WSL-UML notations that would be of most use when reengineering a batch-oriented system would be the Final State, Guard (in order to manage the control logic within the system), State, Call and possibly Signal Event, SubMachine (used to model procedures), and Transition. Consequently, while extensions to the WSL-UML notation can be used to represent statecharts, statecharts can not be feasibly extracted from a batch system so no examples of statecharts, extracted from the system source code and in WSL-UML notation or in UML visual diagrams, are provided in this thesis.

4.9.5 Interaction Sequence Diagrams

Collaboration diagrams are logically equivalent to sequence diagrams; consequently, collaboration diagrams are not drawn. Collaboration diagrams also depict the interaction of class objects but focus on the role of objects within this interaction while sequence diagrams focus on the role of the interactions themselves (OMG, 2004).

By utilising a control graph of procedure calls and the independent task evaluation algorithm, in particular the task granularity of a procedure, it is possible to construct a sequence diagram of procedure calls. These procedure calls may be within their own enclosing object or they may invoke procedures that are enclosed by different objects. If the procedure calls are inter-object calls, the sequence diagrams show them as such.

The event structure as defined in the statechart diagram can be used for both the sequence and activity diagrams.

More details regarding the methods and algorithms to extract sequence diagrams are available in Chapter 7.

4.9.6 Deployment Diagrams

By parsing the source code, in particular the calling of peripheral devices and databases, within code modules, it is possible to develop a deployment diagram which outlines these devices and the relationship between the software system and them. The deployment diagram will depict the pieces of software that utilise a particular device or database. The deployment diagram is particularly important when the system is moved to a new platform or new devices are installed. An example, without a deployment diagram and without the changes to the software that the deployment diagram indicates should occur in terms of changed device calls, the existing software may still call the non-present devices.

4.9.7 Component Diagrams

By parsing the source code, in particular the calling of libraries and software modules, it is possible to develop a component diagram that outlines all of the libraries, modules, et al of the system and the relationships among them. These relationships may include which software module will call another software module. These deployment diagrams are particularly important in large legacy systems which, for easier modification, have been broken down into hundreds of software components, but due to incremental software maintenance, these relationships have been forgotten.

ComponentsList: list of Components (software libraries, copybooks, etc) in a comma-delimited form.

More details regarding the methods and algorithms to extract component/deployment diagrams are available in Chapter 9.

4.9.8 Use Case Diagrams

Use Case diagrams are outside the scope of the thesis because their extraction involve expert user intervention and knowledge. An example, the extend and Include associations, while very useful when reusing Use Cases during business analysis, are not very useful in reengineering because, supposing after the use cases have been extracted from the source code, expert user intervention in their specification would be required.

4.9.9 Object Diagrams

The thesis' reengineering effort focuses on the automated extraction of UML diagrams from batch-oriented legacy systems using source code as the only existing software artefact. Within these constraints, a one class per object is a practical approach. Without user input, aggregating classes into any type of hierarchy, unless it is an obvious hierarchy such as file objects linked as sub-classes to a generic FILE super class, is difficult. Pattern matching and analysis of couplings among classes in order to organise the identified objects into a hierarchy of classes, without user input, is prone to error. Consequently, automated aggregation of identified objects into hierarchies is not performed; instead, for each identified object, a class is created.

Also, because TAGDUR does not have a sophisticated garbage identification mechanism, which would detect both when an object is needed and when the object is no longer needed, objects are created at the program start and are destroyed at the program termination.

Given the fact that the reengineering effort produces a one-object, one-class representation and object lifelines are all the same (the duration of the program), the distinguishing features of an object diagram, multiple instances (objects) of the same class and different object lifelines, are missing. Consequently, the class and object diagrams are very similar.

4.10 UML Components Expressed as WSL Constructs

In order to express UML diagrams in a formal notation, WSL is extended, in this thesis, to map to various UML modelling notations. Consequently, WSL is extended to represent high-level abstract modelling concepts and elements. The UML diagrams produced by TAGDUR are written in a WSL notation that maps syntactically to their corresponding diagram structures in the UML 1.5 Specification (OMG, 2004). By incorporating the latest fully available version of UML, version 1.5, into its WSL diagrammatic notation, TAGDUR ensures that its diagrammatic notation correspond to UML's most currently released specification. WSL is extended to represent high-level constructs such as UML's classes, associations, and activities. By keeping this diagrammatic notation within WSL, TAGDUR maintains the advantages of the WSL representation of the system, notably tool independence, but because the WSL and UML diagrammatic notation are syntactically similar, conversion from this WSL notation into any UML exchange format, such as XMI, and hence, their importation into many UML graphical tools may be accomplished. Furthermore, through WSL, there is a direct relation between the original source code, the model of the restructured system, and the reengineered target system. The WSL-UML notation that is extracted from the source code is

converted into XMI and then imported into a UML visual modeling tool that produces visual UML diagrams of the system (see Appendix C).

All of the WSL constructs, which are used to represent UML components, make use of two WSL features. One feature is the WSL STRUCT or structure which is a collection of contiguous elements. A STRUCT is delimited by the BEGIN keyword, which indicates the beginning of the collection of elements within the structure, and the END keyword, which indicates the end of this element collection. The other feature is the WSL list that is a sequential collection of related items. This WSL list may consist of single elements, ordered pairs of elements, sub-lists, or collections of other elements. A comma delimits each element in the list.

Each WSL construct that represents a UML component, such as an action state, has a unique name. A diagram, such as an activity diagram, consists of a graph of these constructs linked together by their unique names.

UML specification 1.5 was used to model the equivalent WSL constructs. The reason for the selection of version 1.5 was that, during the period of time that this thesis was finalised at the end of 2004, the latest version of UML, version 2.0, had not formally been released. Consequently, this UML to WSL mapping is done with the most recent released UML version.

The WSL constructs form the most-commonly-used elements within the UML components.

A list of the UML –WSL diagram components is as follows:

View	UML Diagram	UML Component	WSL Construct	Used in Thesis
Business Process	Use Case	Use Case	Use Case	No
Business Process	Use Case	Actor	Actor	No
Business Process	Use Case	-	UseCaseList	No
Business Process	Use Case	UseCaseExtend	UseCaseExtend	No
Static	Class Diagram	Package	Package	Yes
Static	Class Diagram	-	PackageList	Yes
Static	Class Diagram	Comment	Comment	Yes
Static	Class Diagram	CommentLink	CommentLink	Yes
Static	Class Diagram	Class	Class	Yes
Static	Class Diagram	ParameterList	ParameterList	Yes
Static	Class Diagram	ProcedureList	ProcedureList	Yes
Static	Class Diagram	-	AttributesList	Yes
Static	Class Diagram	Association	Association	Yes
Static	Class Diagram	AssociationEnd	AssociationEnd	Yes
Dynamic	Object Diagram	Object	Object	No
Dynamic	Object Diagram	Package	Package	No

Dynamic	Object Diagram	-	PackageList	No
Dynamic	Object Diagram	Comment	Comment	No
Dynamic	Object Diagram	CommentLink	CommentLink	No
Dynamic	Object Diagram	ParameterList	ParameterList	No
Dynamic	Object Diagram	ProcedureList	ProcedureList	No
Dynamic	Object Diagram	-	AttributesList	No
Dynamic	Object Diagram	Association	Association	No
Dynamic	Object Diagram	AssociationEnd	AssociationEnd	No
Dynamic	Statechart	State	State	No
Dynamic	Statechart	Transition	Transition	No
Dynamic	Statechart	Guard	Guard	No
Dynamic	Statechart	SignalEvent	SignalEvent	No
Dynamic	Statechart	StateMachine	StateMachine	No
Dynamic	Statechart	SubmachineState	SubmachineState	No
Dynamic	Activity Diagram	Activity	Activity	Yes
Dynamic	Activity Diagram	ActivityState	ActivityState	Yes
Dynamic	Activity Diagram	Transition	Transition	Yes
Dynamic	Sequence and Collaboration Diagram	Sequence	Sequence	Yes
Dynamic	Sequence and Collaboration Diagram	Message	Message	Yes
Dynamic	Sequence and Collaboration Diagram	Partition	Partition	Yes
Dynamic	Sequence and Collaboration Diagram	_	PartitionList	Yes
Architectural	Component Diagram	Component	Component	Yes
Architectural	Component Diagram	ComponentDeploymentLocation		Yes
Architectural	Component Diagram	Resident	Resident	Yes
Architectural	Component Diagram	Implementation	Implementation	Yes
Architectural	Component Diagram	-	ComponentList	Yes
Architectural	Deployment Diagram	Deployment	Deployment	Yes
Architectural	Deployment Diagram	DeploymentLink	DeploymentLink	Yes

Architectural	Deployment Diagram	_	DeploymentList	Yes

Table 4.10.0.1 List of UML Components – WSL Construct Mappings

4.10.1 Example of UML to WSL Notation Mapping

A small example of a class diagram is given below. This diagram consists of two classes, Person and Customer, and has an association of one-to-many between them because one Person may represent several Customers to a company.

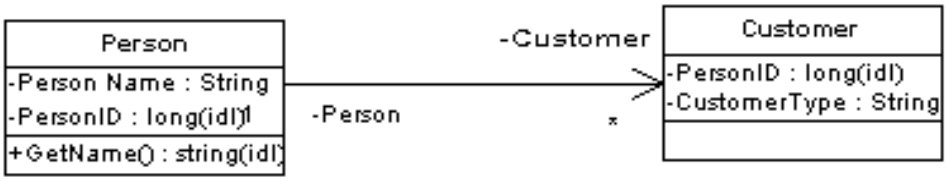


Fig. 4.10.1.1 An Example of a Class Diagram with two classes and a one-to-many association between them.

Each package consists of a set of classes within a diagram. Hence, the WSL notation of package, defined as a structure, would consist of both a package name and a class name list. The WSL notation of class, also defined as a structure, consists of a class name, a visibility name, an attribute list, and an operations list. An attribute list is a list of ordered pairs in the format (VisibilityName: AttributeName). VisibilityName can be one of the following visibility options: + public, # protected, - private, and ~package. An operations list consists of a list in the format <VisibilityName><OperationName> (<ParameterList) : <ReturnTypeExpression><PropertyString>. <OperationName> is the name of the Operations; <ReturnTypeExpression> is an optional expression that describes what an operation returns. <Property String> indicates property values that apply to the element. In the case of Operations, these property values may include <Query> which, if true, indicates an operation that does not modify the system state and <Concurrency> which has the following values: sequential, concurrent, or guarded (the operation's invocation is governed by a guard condition). The parameter list is in the format: <KindName><ParameterName> :<Type-Expression> = <Default_Value>. <KindName> indicates the parameter direction such as in, out, or inout. <Type-Expression> is the data type of the parameter. The Association consists of a structure including the fields of AssociationName, MultiplicityName, AssociationEndName1, and AssociationEndName2. The AssociationEnd structure has the fields of Ordering (whose value can be ordered or unordered), Navigability (whether an association is navigable or not), AggregationIndicator (if yes, indicates that the association is an aggregation association between classes), RoleName, Changeability (if this association is changeable or not), and VisibilityName. Multiplicity may be a one-to-one [1..1], one-to-many [1..*], zero-to-one[0..1], zero-to-many [0..*], and many-to-many [*..*] (OMG, 2004). Because WSL is typeless, each WSL structure has an additional field ElementType that indicates what UML element this structure represents.

In the above example, the WSL notation would be as follows:

Var Struct ClassPackage

Begin

Var ElementType := Package

Var PackageName := Package

Var ClassList := Person, Customer

Var AssociationList := PersonCustomerAssociation

End

Var Struct Person

Begin

Var ElementType := Class

Var AttributeList := {Private, PersonID},{Private, PersonName}

Var OperationsList := Public GetName():String

End

Var Struct Customer

Begin

Var ElementType := Class

Var AttributeList := {Private, PersonID}, {Private, CustomerType}

End

Var Struct PersonCustomerAssociation

Begin

Var ElementType := Association

Var AssociationEndName1 := PersonAssociationEnd

Var AssociationEndName2 := CustomerAssociationEnd

End

Var Struct PersonAssociationEnd

Begin

Var ElementType := AssociationEnd

Var AssociationEndName := Person // name of linked class object

Var Ordering := unordered

Var Navigability := yes

Var AggregationIndicator := none

Var RoleName := 'No Role'

Var Changeability := Yes

Var Visibility := Public

End

Var Struct CustomerAssociationEnd

Begin

Var ElementType := AssociationEnd

Var AssociationEndName := Customer // name of linked class object

Var Ordering := unordered

Var Navigability := yes

Var AggregationIndicator := none

Var RoleName := 'No Role'

Var Changeability := Yes

Var Visibility := Public

Var Multiplicity := [1..*] // put on the target end of an association end only to denote multiplicity of the association

End

The purpose of this UML to WSL structure mapping is two-fold. One is to extend WSL to represent high-level abstract modelling concepts and elements and the other is to make this mapping UML-compliant such that these structures can be exported, via XMI, to selected visual UML tools.

UML has a topmost, metamodel layer which serves to unify its various diagrams and also serves to provide extensions to the UML model. The Meta-Model consists of several logical packages: Foundation, Behavioral Elements, and Model Management. The Foundation package, in turn, consists of the Core, Extension Mechanisms, and Data Type sub packages (OMG, 2004: pp 2-11, 2-13). In reengineering, there is often no need to use the Extension Mechanisms of UML to represent a system. Typically, one would try and confine the reengineered and re-documented system within the common and already-defined set of diagrams. In this reengineering and re-documentation effort, since COBOL is converted from its data-typed source code into a data type-less WSL intermediate representation, there is no need to express this documentation using the Data Type sub package. Because the generated re-documentation includes the derivation of behavioral diagrams such as those of activity and sequence from source code, the Behavioral Elements meta-model of UML is utilised. Model Management deals with the grouping and organisation of various model elements into Models, Packages, subsystems, and UML profiles (OMG, 2004: pp 2-181, 2-184).

In order to demonstrate that this WSL-UML mapping is semantically equivalent, references to this Core Package and the UML model elements that are defined in WSL are made when necessary. An example, a class model element in the class diagram of the UML has a reference to its meta-model element, Classifier, to ensure that the mapping of this WSL notation to the UML is not only to a specific diagrammatic element but to the UML Core Package itself.

4.11 Summary

The purpose of this chapter is to outline the various processes that a selected legacy system will undergo in order for a series of UML diagrams to be extracted satisfactorily from it. These UML diagrams, in turn, will help re-document the restructured system and enable developers to gain a better understanding of the current system.

This chapter first describes the selected legacy system and provides both the business and academic cases for its restructuring and re-documentation. The first step in this re-documentation and restructuring phase is to convert the original source code into a formal intermediate representation, in this case WSL. The chapter then outlines the background of the various restructuring and UML diagram extraction processes. Further details, including algorithms, regarding these processes are provided in later chapters. This thesis focuses on a particular legacy system, a batch-oriented COBOL system, in order to determine if it can be satisfactorily translated into a formal based intermediate representation of WSL and then restructured into an object-oriented, event-driven system. Each step of this restructuring process must be validated in some way to ensure that these processes do not produce inherent errors in the restructured system. Once the system has been restructured, various methods are used in order to determine if a particular UML diagram can be extracted and the reasons for the success or failure of their extraction.If a diagram can be satisfactorily extracted, algorithms are formulated for their extraction from the source code. A combination of a formal notation, WSL, with a software visualisation modelling notation, UML, was introduced, as WSL-UML notation, in order to model UML modelling concepts, such as activities, into their semantic equivalents in WSL. This WSL-UML notation is then converted into visual UML diagrams. Finally, the extracted UML diagrams are validated, through a validation checklist, to ensure that they accurate represent the restructured code. The purpose of this approach is to determine if validated UML diagrams may be extracted from a legacy system and if this extraction occurs, under what conditions and using what algorithms, in order to gain a better system understanding for the developer.

Chapter 5: Converting COBOL to WSL

5.1 Introduction

In order to reengineer a legacy system in COBOL through the specified reengineering process, this COBOL program must first be translated into WSL. The translation from the legacy system's source programming language to WSL provides a programming-language independence to the later transformation and code generation phases. In this chapter, several technical issues are outlined in regards to COBOL to WSL conversion; in addition, various attempts that were used to create parsers to handle the COBOL to WSL translation phase and the reasons for their failure are outlined. Because of these various failed attempts, it was decided to convert this system manually from COBOL to WSL. A list of rules for this manual conversion is outlined in the chapter.

5.2 Technical Issues

Van den Brand argues that in order to reengineer software, it is first crucial to have tool support and then to have these tools based on the grammar of the language that they intend to process. The argument given is that the tools, by having a strong basis on the language of their source system, are better structured to derive meaning from this system and reduce the number of false-positives that a more generic tool might produce. An example, a generic tool might not recognize a comment because the comment syntax is specific to a particular language and hence, might include the comment in its analysis and pattern matching (van den Brand, 1997).

Van den Brand observed that parsers first depend on syntactically correct programs to parse and secondly depend on a complete set of grammar rules. In COBOL, although efforts were made to define many of the grammar rules of COBOL 85, a huge language with over a thousand production rules, this effort was incomplete (Mammel, 1996). Van den Brand had tried to re-define the syntax within COBOL programs into a more universal, simple program syntax through ASD+SDF Meta-Environment. However, this attempt was not a trivial task for a number of reasons. The ANSI document that describes the COBOL standard is huge. This document is not very well structured with no clear distinction between syntatic and semantic rules. The rules provide the general format that give the formal definition of the syntax, the syntax rules that add properties to the definitions in general format, and the general rules that define the semantics. The arbitrary order of option constructs is often defined informally in the syntax rules rather than the general format (van den Brand, 1997c). Few programs, however, are written in pure COBOL 85. Most programs use implementation extensions, such as extensions to handle database or file access which are proprietary (Fergen, 1994). In a batch-oriented system, file access extensions are used often more extensively than other types of legacy systems. Batch-oriented systems depend on files for their input/output more extensively than more user-oriented types of systems.

There are a myriad of COBOL dialects which presents problems in both defining a syntax for a common COBOL program and using available tools. Lex + YACC tools are commonly used scanner and parser generator tools. However, LEX + YACC are based on LR-parsing which, because of its context-free grammar nature, does not handle shift-reduce and reduce-reduce conflicts well. This problem is particularly evident in COBOL with its associativity and priority conflicts that are inherent in the rules of

some of its grammar dialects. A COBOL-specific example would be, X = Y or Z, which depends on the status of Z to determine if the root of the parse tree is either = or OR. Another problem with COBOL is that there are many syntactic variations with exactly the same syntax (van den Brand, 1997).

5.3 Parsing COBOL Efforts

Although many parsers are based on the premise that the grammar of the language is known and this grammar is unlikely to change constantly, this premise is not the case for COBOL where there is a myriad of COBOL dialects with extensions to COBOL such as embedded CICS, SQL, and/or DB2. Since the language of COBOL is not well-known enough to satisfy the parser's requirements, this parser is not satisfactory in the field of reengineering. The ambiguity of COBOL results in several shift/reduce conflicts – it is not unusual that a single COBOL grammar rule addition results in 50 shift/reduce conflicts (van den Brand, 1998).

Lex and Yacc are well-known method tools to generate scanners and parsers. Although they have been used to generate COBOL grammars, these tools are LR which make COBOL grammars difficult to extend with no tool support for locating conflicts. TXL is a program transformation system, built on LL(1) with backtracking. With no depth on backtracking, unexpected parse trees can be generated which can not be handled. TAMPR is a general purpose program transformation system that has been used to restructure COBOL source. One of TAMPR's authors, McParland, admits that this system has problems in dealing with the various dialects of COBOL. Other parsers for SES Software-Engineering Service GmbH are hand-crafted rather than developed using parsing tools such Lex or Yacc because these tools, according to SES' technical director Harold Sneed, do not serve their purposes. Sneed also encountered difficulties with the various dialects of COBOL, which a hand-crafted parser could be modified to handle (van den Brand, 1998).

Another problem occurs when home-grown pre-processors are used in conjunction with a COBOL parser. An example, when parsing a OS/VS COBOL, the parser did not recognize some COBOL programs. Upon further investigation, they found ampersands where an identifier was expected in 40% of these cases. These ampersands were not part of any COBOL grammar. These ampersands were used as parameter mechanisms for generating programs by this home-grown pre-processor (van den Brand, 1998).

Converting COBOL to a formal denotational semantics, such as WSL, has been accomplished by others such as Baumann. Baumann analysed several common COBOL constructs, such as goto statements, perform statements, paragraphs, etc, and described the semantics of these constructs formally. These definitions, in turn, were expressed in their programming language, Mico, which was implemented in ML. The purpose of these formal semantics was to enable the sound program analysis and semantic invariant program transformations (Baumann, 1993).

The large number of COBOL production rules, plus the many dialects of COBOL, have an effect on the completeness and on the feasibility of using available COBOL compilers in converting a COBOL program to another language. There is a freeware COBOL RPG compiler on the Internet, but it is not a complete compiler (Huggins, 2004). Chris Verhoef and Ralf Lammel tried to extract the VS COBOL II grammar from IBM's reference manual in order to provide a publicly-available COBOL grammar. However, this reference manual contained many errors and the resulting COBOL grammar that was extracted was still

ambiguous (Lammel, 2001). TinyCOBOL is a project to produce a COBOL parser based on the COBOL 85 grammar but this tool both does not strictly adhere to the COBOL 85 standard and has limited functionality (Huggins, 2004). CobCy was an attempt to convert COBOL to C, for easier translation. CobCy was never completed; it is now being reworked to convert COBOL to Java with the tool's status is still incomplete (Santini, 2004).

Kwiatkowski tried to create a parser, Kinga, to convert COBOL to WSL. One problem with Kwiatkowski's approach was that this WSL record definition report mirrors that of a COBOL record structure, complete with "redefines", which is very difficult to convert into a standardized record structure. Furthermore, the selected sample of COBOL code that he converted to WSL uses keyboard and monitor input and output, rather than file systems. File systems are used extensively in this batch system and COBOL file systems come with their own set of difficulties in conversion such as indexes, positional rather than name record field accesses, and implicit type conversion of record fields from one record to another (Kwiatkowski, 1998). Kinga was used to parse the selected COBOL system but this tool crashed, with a irrecoverable parsing error, when it tried to parse the COBOL samples.

An attempt was to use GOLD's parser, with Lammel's extracted COBOL grammar, to construct a COBOL parser that would be able to parse the selected legacy system (Gold, 2004). However, Lammel's extracted grammar did not match the dialect of the selected COBOL system and again, the GOLD parser failed with an irrecoverable parser error. A similar attempt was made to parse the selected legacy system using the COBOLTransformer Toolkit, which contains a COBOL parser, but again this parser failed to parse the source code of the selected legacy system and failed with an irrecoverable parser error (Siber, 2004). Faced with the inherent difficulties of creating a COBOL to WSL parser and the failure of several available tools to parse the COBOL code of the selected legacy system, it was decided that it would be better to manually convert the COBOL code into WSL rather than rely on some parser/analyzer tool.

5.4 Restructuring COBOL Code

The original code may contain unstructured statements such as "GO TO" lines in programs written in COBOL language that make the source code become "spaghetti code" (Hutty, 1997). Because COBOL systems often have unstructured source code or sometimes a mixture of structured and unstructured source code, the first task is to restructure the legacy system into structured COBOL. A number of algorithms are available to eliminate these GOTOS and convert unstructured COBOL to structured code (Klosch, 1996). By restructuring, dangling GOTOs are eliminated and the programs are sectioned off into groups of distinct procedures. This selected legacy system has already been restructured into structured COBOL; consequently, these restructuring algorithms were not used.

5.5 Description of COBOL Programs

Once the COBOL is restructured if necessary, the various divisions of the COBOL program are identified. COBOL programs typically have four divisions:

1) Identification – identifies the program.

2) Environment – describes the hardware platform plus supporting hardware to be used by the program.

3) Data – defines the data structures and characteristics of the data to be used within the program. This data includes both file and non-file data.

4) Procedure – contains the set of instructions and program logic of the program (Feingold, 1981: p 61).

Although the program code within the Identification and Environment divisions is parsed and the information contained within this code, such as the name of the program or specified hardware platform, is stored and used for purely commentary purposes in the documentation phase of this reengineering, the program code within these divisions is not translated to WSL. The name of the program may be used in the case of a COPY command in order to identify each section of code as contained within a program file but this program identification is not translated to WSL code directly.

The two remaining divisions within the COBOL program, the Data and Procedure Division, are of primary use to this reengineering process. The Data division contains the file descriptions and variable declarations that are directly translated to WSL.

5.6 Data Division

5.6.1 File Section

5.6.1.1 Theory of Files in WSL

According to Martin Ward (Ward, 1992), a table in WSL is defined as a partial function that maps a name (the key) to the data stored under that key. This is used to record information about the item. A table is valid if the information within it is correct; an example, if the table contains a pair of the form (fn-name, S) then evaluating the function fn-name on that item should return S as the result.

$$\text{Tables} = (\text{Names} \rightarrow\!\!\!\!\!\!\mid\ \ \text{Data})$$

Tables consist of Names (keys) mapped to Data items.

Because some of the query functions take a relative position (Posns) as a second argument, the following format is used:

> Tables consist of (Names (keys) mapped to Data items) Union (Names [keys] cross-referencing Positions [relative positions] mapped to Data items)

Using this definition of a table and Liu's definition of reading and writing from a physical device (Liu, 1999), separate read and write constructs in WSL can be defined. Liu and Millham define an input statement in WSL as the following:

> Files, in WSL, can be envisioned as list of elements where the elements are records. These elements, in turn, may have sub-lists or sub-elements. Access to these elements is via an index value where the index value is a positive integer between 1 and N where N represents the last element of a list. Index, used in Put and Fetch commands, is used to indicate a specific element in a list of elements. It is assumed that a physical shunt to be a list of elements of 1..n. Each element is of uniform size and represents the granular unit of I/O that is being read or written during an I/O operation. The index indicates the specific element of this array that is being accessed. (Liu, 1999)

In this way, shunts, which represent file I/O in WSL, act upon sequential lists of elements, or records. These records are referred to by an index pointer, whose value denotes their sequential position in the list from 1 to last element N. In practical terms, it means that to perform I/O on files in WSL, each record must be referenced, whether in reading or writing of a file, with its index value.

Random-disk and indexed files, although it is possible to represent these characteristics in WSL, are out of scope of this thesis. Hence, information such as the file characteristics and its index are recorded for documentation purposes and for later use during the planned data reengineering phase.

5.6.1.2 Files in WSL

Each file has its own implicit structure contained within it. A file, F, has an implicit structure with record field such as ISOPEN, which is a Boolean field that indicates if the file is open or not, and DIRECTION, which is a text field that indicates if the file is being opened for input./output/or both. Opening a file means that the associated file structure's field, ISOPEN, is set to "Y". A file opened for input has its file structure's DIRECTION set to "INPUT". The setting of these file structure's fields are for documentation/representation purposes only; in this way, the platform independence of WSL is maintained. Many operating systems use stateless files, which do not require files to be opened, while other operating systems require the use of state files (Tannenbaum, 1992).

However, some features of COBOL make this tracking of data flow between different sub-systems difficult. One feature is that a file, represented by a physical device, can be referred to through many different aliases; in fact, it can be accessed using a different alias for each subsystem accessing it. Thus, there is no unique nor single point of reference in COBOL file accesses to indicate if a file that is being used by one subsystem, using one alias, is the same as the file used by another sub-system, using another alias. One way around this use of aliases is to use the file's physical device name. Thus, even though a program may refer to the same file using different aliases, the use of the file's physical device name provides a single point of reference to indicate that all sub-systems are accessing the same file (Feingold, 1998).

Another specific feature of COBOL is that it can be a typeless language when dealing with records. COBOL depends on the size and length of fields within a record in order to demark fields rather than any specific data types. An example, a record, Person, with fields Name of string of size 30, Date-Of-Birth of date of size 6, and Salary of type decimal and of size 8 may be referred to in different ways in a COBOL record. An example, one subsystem may define this record of type String and Size 44 while another subsystem may refer to the same record as Name of string of size 30 and Salary of decimal of size 8 but decompose the 6-digit date field into 2-character strings of Year, Month, and Day. This lack of uniformity makes the use of detecting data dependencies between different records based on the size, type, and position of their fields impractical, even though this detection practice has been advocated by some.

COBOL relies on data size and position rather than element matching between corresponding columns of different records. An example, the following file record has no named columns:

FILE-REC of size 100 : in WSL, STRUCT FILE-REC

While another data records may have several columns of the different names and sizes

DATA-REC of total column size 100 : in WSL,

```
Struct DATA-REC
Begin
  NAME  /* size 50 * /
  ADDRESS  /* size 25 */
  CITY /* size 25 */
End
```

COBOL may, in a file record assignment, imply a certain record structure in FILE-REC similar to that of DATA-REC. An example, the COBOL statement:

```
MOVE DATA-REC TO FILE-REC
```

Implies the following constructs in WSL:

```
FILE-REC.ELEMENT1=DATA-REC.NAME
FILE-REC.ELEMENT2=DATA-REC.ADDRESS
FILE-REC.ELEMENT3=DATA-REC.CITY
```

Although FILE-REC has no named columns within its structure, COBOL assumes that it would have a similar structure of derived columns with each column having the same starting position and size as the columns of the DATA-REC. The derived columns of this implicit record structure follow the naming convention of ELEMENTn where n is an integer from 1 to infinity and where each successive element is named ELEMENT(n+1) of its preceding element ELEMENT(n).

Because WSL is a typeless language, the data size and data type of each variable along with the implicit columns of a record must be kept in a separate database rather than appear within the WSL code itself.

An important note is that this program reengineering does not do any error-checking of the original source code. Thus, if in one place, the code assigns FILE-REC.ELEMENT2 a variable of size 20 and of type string and then, in another places, tries to assign the same column a variable of size 30 and of type numeric, no automatic error detection nor handling will be invoked. This is partially due to the typeless and sizeless nature of WSL and partially because error detection in the original source code is beyond the scope of this thesis.

Because one of the peculiarities of COBOL is the ability to transfer a record structure from one record variable to another without the need to transfer explicitly their constituent record fields, care must be taken to ensure that whenever a record-level transfer of data from one variable to another occurs in the COBOL code, all constituent record fields of the record structure in question are explicitly named and transferred. The Data Dictionary, formed from the Data Division of COBOL, is used to identify these record-level variables and their constituent record fields for this purpose.

5.6.1.3 File Management

In the following COBOL statement, SELECT 210-PENDRPT-FILE ASSIGN TO PENDRPT, the file 210-PENDRPT-FILE is assigned to a physical device, PENDRPT. Since files often are used both as both data stores and as interfaces between COBOL programs or copybooks, it is important to determine the non-aliased connection of data between these programs. Thus, all file variables, regardless of file aliases, should refer to their actual physical filename or device and that this physical filename or device should be plainly depicted as a component of a class which interfaces between the internal classes of different copybooks. Many programming languages allow logical names to be used as an alias for a physical device or filename. In order to handle this aliasing, a new WSL construct, *PHYS_DEV*, is defined. *PHYS_DEV* is an extension to the logical file name record structure which indicates the physical file name that corresponds to the logical file name in the format :

'SELECT' <LOGICAL_FILENAME> 'ASSIGN' 'TO' <PHYSICALDEVICE> **::=** <LOGICAL_FILENAME>' '.PHYS_DEV' ':=' <PHYSICALDEVICE>

To assign a record name to a logical file, the following production rules are followed:

<FILENAME> 'DATA RECORD IS '<FILERECORD>'.'**::=** <FILENAME>.DATRECORD := <FILERECORD>

Handling the opening and closing of files are managed through the following production rules:

'OPEN' 'INPUT' <FILENAME> **::=** <FILENAME>'.ISOPEN' ':=' 'TRUE' ';' <FILENAME>'.DIRECTION' ':=' 'INPUT';' <FILENAME>'.INDEX' ':=' '0'

'OPEN' 'OUTPUT' <FILENAME> **::=** <FILENAME>'.ISOPEN' ':=' 'TRUE' ';' <FILENAME>'.DIRECTION' ':=' 'OUTPUT' ';' <FILENAME>'.INDEX' ':=' '0'

'OPEN' 'INPUT-OUTPUT' <FILENAME> **::=** <FILENAME>'.ISOPEN' ':=' 'TRUE' ';' <FILENAME>'.DIRECTION' ':=' 'INPUT-OUTPUT';' <FILENAME>'.INDEX' ':=' '0'

'OPEN' 'OUTPUT' <FILENAME> 'FOR' 'APPEND' **::=** <FILENAME>'.ISOPEN' ':=' 'TRUE' ';' <FILENAME>'.DIRECTION' ':=' 'INPUT-OUTPUT';' <FILENAME>'.INDEX' ':=' <LAST-RECORD-INDEX>

'CLOSE' <FILENAME> ::= <FILENAME>'.OPEN' ':=' 'FALSE'

Each file is associated with a file record which, in turn, is associated with a file index which indicates which current record is being accessed.

Often, COBOL files contain additional information which is captured within the File STRUCT but serve no purpose within the UML diagram generation. This information includes information such as the type of records and the number of records per block.

'LABEL' RECORDS' 'ARE' 'STANDARD' ::= <FileName>'.LABEL RECORDS' ':=' 'STANDARD'

'BLOCK' CONTAINS' <VALUE> 'RECORDS' ::= <FileName>'.BLOCK_OF_RECORDS' ':=' <Value>

Associating a record with a file is accomplished through the production rule:

'DATA RECORD' 'IS' <RECORD-NAME> **::=** <FILE>.DATA_RECORD := <FILE_RECORD_NAME>

Because WSL does not yet possess well-defined file system characteristics, much of the platform-specific file characteristics of the COBOL program are retained for purely documentation purposes but are not directly translated to WSL. Thus, the description of the file system as indexed, sequential, et al has no direct representation within WSL at present. WSL represents file systems as a simple physical shunt. Although WSL could be further developed to represent complex file systems, this representation has the potential of making the WSL file system definitions platform-dependent.

5.6.1.4 Reading, Writing Using Files and Index Arithmetic

5.6.4.4.1 File Input

File input (or file record reading), in WSL, is handled through the WSL construct:

Fetch *x,y,z*

where *x* is the variable destined to hold the value retrieved from shunt *y*, *y* is the physical shunt, and *z* is the key mapped to the data item in shunt *y* that is to be to be retrieved, and *s* is a shunt or physical device. If there is no mapping of keys to data in shunt s, y is an optional parameter which may either be left out or set to 0.

Handling a file read with a value assignment at the end, the following production rule is used:

'READ' <FileRecord> 'AT' 'END' 'MOVE' <VALUE> 'TO' <VARNAME> **::=** 'FETCH' <DESTINATION-RECORD> ',' <SOURCE-FILE-RECORD> ',' <SOURCE-FILE-RECORD-INDEX> ';' 'IF' <SOURCE-FILE>'.EOF' 'THEN' <VARNAME> ':=' <VALUE> ';' 'FI'

Assignments of record fields are done using a positional rather than a nominal basis. An example, record field R1 of record Rec1 begins at position 45 in the record and has a length of 4 bytes. Assignment of record field, R1, to another record field, R2, of record R2 must occur at the same position and must have the same length.

5.6.4.4.2 File Output

File output (or file record writing), in WSL, is handled through the WSL construct:

Put *x,y,z*

where *y* is the variable holding the value to be written to shunt *x*, *z* is the key mapped to the data item in shunt *x* that is to be to be retrieved, and *x* is a shunt or physical device. If there is no mapping of keys to data items in shunt *x*, then *z* is an optional parameter which may either be left out or set to 0.

Mapping of an element to its key is a language-independent feature in that the key corresponds to an element's identity or position within a set. An example, for set {e1, e2, … en}, the key k_i represents a means to identify element e_i either through a unique key identity value or through its $position_i$ in the set; how the physical program implements this mapping, either as a record offset in the case of a sequential file or as a key-physical mapping in the case of random-access files, is not relevant to us.

5.6.4.4.3 Reading and Writing Printer Records

Handling printer records are handled using the following production rules:

'WRITE' <PRINTER-RECORD> 'BEFORE' 'ADVANCING' 'BY' <VALUE> 'SPACES' **::=** 'WSLLoopCnt' ':=' '1' ';' 'WHILE' 'WSLLoopCnt' '<=' <Value> 'DO' 'WRITE' ' ';' 'OD' 'WRITE' <PRINTER-RECORD>

'WRITE' <PRINTER-RECORD> 'AFTER' 'ADVANCING' 'BY' <VALUE> 'SPACES' **::=** 'WRITE' <Printer-Record> ';' 'WSLLoopCnt' ':=' '1' ';' 'WHILE' 'WSLLoopCnt' '<=' <Value> 'DO' 'WRITE' ' ' ';' 'OD'

5.6.4.4.4 Index Arithmetic

Index arithmetic is handled using the following production rules:

'SET' <FILE-RECORD-INDEX> 'TO' '1' **::=** <FILE-RECORD-INDEX> ':=' '1'

'SET' <FILE-RECORD-INDEX> 'UP' 'BY' <VALUE> **::=** <FILE-RECORD-INDEX> ':=' <FILE-RECORD-INDEX> + <VALUE>

'SET' <FILE-RECORD-INDEX> 'DOWN' 'BY' <VALUE> **::=** <FILE-RECORD-INDEX> ':=' <FILE-RECORD-INDEX> '-' <VALUE>

5.6.3 Working Storage Section: Variables in COBOL

A typical variable declaration in a COBOL program has the following format:

<level_number> <variable_name> <data_type_and_size> <initial_value>

The <level_number> indicates whether the variable, within this declaration, is a file, record name, record field, or independent variable. A level number of "01" indicates a file variable, in particular a record name used by the files within the program. A level number of 05 indicates either a record name, in the case of the immediately succeeding variables declared below it having a level number of less than "05", or an independent variable, in the case of the immediately succeeding variables declared below it having an equal or greater level number.

Each variable has an implicit record structure of

1) Type
2) Size

3) Format_prefix

4) Space_separator

5) Decimal_separator

6) Positive_negative_sign

One particular aspect of COBOL is the use of numerals at the start of a variable name. An example, a COBOL variable name may be "1000-READ-COUNT". Similarly, many procedural names in a COBOL program may begin with a number. The numeric starting character of a variable name is illegal in many languages such as C++. The variable names as declared within the COBOL program are retained as they originally were declared within COBOL. WSL, unlike C++, has no restrictions as to its variable names. Consequently, the original COBOL variable names are retained as the program is transformed and then documented. It is important to retain as much of the original variable or procedure names of the COBOL program as possible because often these variable names contain valuable domain knowledge (Pu, 2003).

Because WSL is a type less language, many important characteristics such as a variable's data type and size are retained in the database for use during the C++ code generation phase but are not directly translated to WSL. An example, the COBOL variable declaration "05 000-REC-KEY X(3)" is translated to variable as "VAR 000-REC-KEY" with no designated data type.

5.7 Procedural Division

In order to encapsulate the original COBOL program within procedures, a main calling procedure which contains the main code that calls other procedures and which initialises the values of variables is needed. This procedure is usually denoted by its name in the format of <Program_Name>-Init. All the initial values of COBOL variables as specified by their Value keyword in the format: <level_number> <variable_name> Value <InitialValue>. Sometimes, multiple initial values may be specified. An example, 88 Pick-Is-Valid Values 1,2,3,9 indicates that the variable, Pick-Is-Valid, has a list of valid values 1,2,3, and 9. This information is stored in the database for documentation and code re-engineering efforts but is not translated directly to WSL. One reason is that variables in WSL can not hold multiple values within one variable simultaneously.

COBOL copybooks, because they often encapsulate a common functionality and because the only in-source documentation occurs at the beginning of the copybook rather than interspersed between relevant parts of the code, play an important part in program understanding. Thus, while it is desired, in the specified data clustering algorithms, to produce classes based on the actual data and control flow of the entire program rather than rely on the current program structure to form the class structure, it is important to retain this copybook structure if only for the logical functional decomposition of the system and for the relevant documentation that it provides.

Thus, it was proposed to encapsulate all of the classes, with their attributes, associations, and methods, originating from the same COBOL copybook into the same package. This package would contain a link to any in-code documentation found at the beginning of the COBOL copybook. The package would also be available as a class grouping mechanism such that the user, by viewing this package diagram, would be able to view the functional decomposition of a program by copybook module. The classes within a package, however, would be free to form their own associations and interfaces with other classes based on actual control and data flow within the program rather than be bound to the existing copybook module structure.

The use of COPYBOOKS in Cobol involves a special conversion algorithm to convert the COBOL COPYBOOK into an integrated WSL program. An example, the COBOL statement, Copy "FileName", involves splitting the COBOL <FileName> file into Variable Declaration and Procedures sections. These sections are then translated into WSL. The Variable Declaration section is then appended to the Variable Declaration part of the current program; while the Procedures section is appended to Procedures part of current program. At present, a special WSL construct, **include**, is used to represent the inclusion of a library or sub-file within its caller program using the rule:

'COPY' <COPYBOOKNAME> ::= 'INCLUDE' <FileName>

5.7.1 Procedure Calls

The Procedure Division of the COBOL code is then parsed. Because a UML representation requires that all code be encapsulated within methods or procedures, all code must be assigned to a procedure and all variables must be scoped accordingly. In the case of the first line(s) of code in the Procedure Division that, normally within COBOL, have no explicit procedure label, an arbitrary procedure label is assigned them, such as Init-Program.

The parsing of the COBOL code then continues. Whenever a procedure label is found, a new procedure is created and all subsequent lines of code are assigned to that procedure until a new procedure label or the end of the code is encountered.

Paragraph Names in COBOL start at positions 8 through 11 and end with a period. Their WSL equivalents are procedures. The end of Procedure Paragraph-Name1 is delimited by the start of another procedure, in this case <Paragraph-Name2>, or the end-of-file.

<ParagraphName1><statements>'.'<End-of-File> || <ParagraphName2> **::=** 'Proc' <ParagraphName1> '(<Parameter VarName>*)' 'BEGIN' <WSL-equivalent statements> 'END.'

The code at the very beginning of the Procedure section, which is not encapsulated within any paragraph and which actually occurs before any paragraphs are declared, is deemed to be the main calling code. This main calling code is encapsulated within a procedure with the name <File_Name>-VAR-INIT. This procedure contains all the assignment statements which give the declared variables their predefined (in COBOL) values and contains the main calling code.

All procedure code is delimited, at its starting position, with a **BEGIN** keyword and at its ending position, with an **END** keyword phrase.

Various COBOL to WSL rules are given in regards to a COBOL statement calling a paragraph (or procedure). In the WSL conversion, any COBOL variables used exclusively within this procedure become the procedure's local variables. Any COBOL variables that are used both outside and inside the procedure must be declared as the procedure parameters. In WSL, **Call** <Procedure Name>, when invoking a procedure, is an optional keyword which is used for easier parsing and syntactic analysis of WSL code.

<VarNameList> ::= <VarName>? || <VarName1>’,’<VarName2>*

‘Perform’ <Procedure Name> **::=** ‘Call’? <Procedure Name>’(‘<VarNameList> ‘)’

‘Perform’ <Procedure Name> ‘Until’ <Condition> **::=** ‘WHILE’ ‘NOT’ <Condition> ‘DO’ ‘CALL’ ? <Procedure Name>’(‘<VarNameList> ’)’ ‘;’ ‘od’

‘CALL’ <Procedure Name> ‘USING’ <VarNameList> ::= ‘CALL’ <Procedure Name> ‘(‘<VarNameList>’)

‘PERFORM’ <Procedure Name> ‘Varying <Loop VarName> ‘FROM’ <Init-VarName> ‘BY’ <Inc-VarName> ‘UNTIL’ <Condition> ::= <Loop VarName> ‘=’ <Init-VarName> ‘;’ ‘DO’ ‘UNTIL’ <Condition> ‘CALL’ <Procedure Name> ‘;’ <Loop VarName> ‘:=’ <Loop VarName> + <Incr VarName> ‘;’ ‘od’

PERFORM <Procedure Name> <Integer-Value> ‘TIMES’ ::= ‘WSLLoopCnt := 1’ ‘;’ ‘DO’ ‘WHILE’ ‘WSLLoopCnt’ ‘<’ <Integer-Value> ‘CALL’ <Procedure Name>’;’ ‘WSLLoopCnt’ ‘:=’ ‘WSLLoopCnt’ ‘+’ ‘1’ ‘;’ ‘OD’

5.7.2 Assignment Statements

Record filler can have multiple elements and sub-elements used in assign or read/write statements. In the case of a COBOL assignment, or MOVE, statement on a bulk basis without individual record fields being specified, record fields are assigned on a positional basis. In other words, if field1 of Record A has an index of 4 from the onset of the record and a length of 4 bytes, it will be assigned to fieldC of Record B which also has an index of 4 from the onset of the record and a length of 4 bytes. If no such correspondence exists, in the case of a record with multiple fields being assigned to a record of 100 characters, each record field of the source record is assigned to a newly-created field of the destination record. These new fields are designated as ElementN where N is the number of the field in ascending order from the beginning of the record. An example, record B will be given fields, element1, element2, and element3, to correspond to an assignment from source record A with fields, field1, field2, and field3.

A move statement without Corresponding keyword:

Move Rec1 into Rec2 uses positional fields so if Rec1.Filler = 128 and is assigned Rec2.element1 and Rec2.element2, then Rec1.Filler.element1:= Rec2.element1 and Rec1.Filler.element2:= Rec2.element2

Move Corresponding <Rec2> to <Rec1> -> moves only those record fields of <Rec2> that have the same name as the record fields of <Rec1> to their matching field in <Rec1> -> <Rec1>.field1:=<Rec2>.field1.

‘MOVE’ <VARNAME1> ‘TO’ <VARNAME2> ::= <VARNAME1> ‘:=’ <VARNAME2>

‘MOVE’ ‘SPACES’ ‘TO’ <VARNAME1> ::= ‘VAR WSLCnt’ ‘;’ ‘DO’ ‘WHILE’ ‘WSLCnt’ ‘<=’ <VARNAME1>’.Size’ ‘DO’ <VARNAME>’.[WSLCnt]’ ‘:=’ ‘ ‘;’ ‘WSLCnt’ ‘:=’ ‘WSLCnt’ ‘+’ ‘1’ ‘;’ ‘OD’

To handle multiple destinations within one COBOL assignment statement in WSL, this single multiple-destination assignment statement is split into multiple WSL assignment statements – one single WSL assignment per destination variable.

‘MOVE’ <Value> TO <VARNAME1>, <VARNAME2>, <VARNAME3>* ::= <VARNAME1> ‘:=’ <Value>’;’ <VARNAME2> ‘:=’ <Value>

5.7.3 Control Constructs

If an "**IF** <*Condition1*> **Then** {*Cobol Statements1*} **Elseif** <*Condition2*> **Then** {*Cobol Statements2*} **Else** {*Cobol Statements3*} statement is encountered, the following production rule is used for conversion into the following WSL equivalent construct.

'IF' <Condition1> 'THEN' <COBOL Statements1> 'ELSEIF' <Condition2> 'THEN' <COBOL Statements2> 'ELSE' <COBOL Statements3> ::= 'IF' <Condition1> 'THEN' <WSL equivalent constructs of COBOL Statements1> 'ELSIF' <Condition2> 'THEN' <WSL equivalent constructs of COBOL Statements2> 'ELSE' <WSL equivalent constructs of COBOL Statements3> 'FI'

WSL denotes the end of the condition block of statements with a "FI" keyword. COBOL denotes the start of the condition block or iteration block (termed a control block) by having statements within this block with a greater left margin than the condition statement as in the example. When the first COBOL statement has a lesser left margin than the COBOL statement(s) within the COBOL control block, this first COBOL statement with a lesser left margin denotes the end of the control block in COBOL. In WSL, statements within a control block are delimited at the start with the **Begin** keyword and at the end with the **End** keyword.

'IF' <Condition1> 'THEN' <Statement1><Statement2>* ::= 'IF' <Condition1> 'THEN' 'BEGIN' <Statement1> ';' <Statement2> ';' 'END'

COBOL uses a variety of keywords and symbols to represent the same operators. An example, IF Account-Balance GREATER THAN 0 THEN is the same as IF Account-Balance > 0 THEN. Operators that are represented by phrases rather than symbols, such as LESS THAN rather than <, are replaced by symbols in WSL.

'IF' <VARNAME> 'GREATER THAN' <Value> 'THEN' ::= 'IF' <VarName> '>' <VALUE> 'THEN'

Another practice in COBOL is to use an if-then-else statement with an empty if condition block rather than negate a condition. An example, IF <Condition> THEN <Empty Statement> ELSE <Else_block_of_Statements>. This <Empty Statement> is usually a null statement such as NEXT SENTENCE. Rather than create an If-Then-Else statement with an empty if block of statements in WSL, this if-then-else statement is converted to a single if statement in WSL by negating the condition in the if statement and having the statements in the else block become the conditional block of the if statement.

'IF' <Condition> 'THEN' <Empty Statement1> 'ELSE' <Statement2> ::= 'IF' 'NOT' <Condition> 'THEN' <Statement2>

A common COBOL statement, PERFORM <Statement(s)> UNTIL <CONDITION>, is translated into an iteration loop in WSL. The WSL equivalent to this COBOL statement is WHILE NOT <CONDITION> DO <Statement(s)>.

'PERFORM' <Statements> 'UNTIL' <CONDITION> **::=** 'WHILE' 'NOT' <CONDITION> 'DO' <Statement(s)>

The COBOL statement, Evaluate <VarName> When <Value>, is translated into multiple if-elif-else statements in WSL such as if <VarName> = <Value>. An example,

'EVALUATE' <VARNAME> 'WHEN' <Value1> 'THRU' <Value2> <Statement(s) ::= 'IF' <VARNAME> '>=' <Value1> 'THEN' <WSL equivalent of Statement(s)> ';' 'IF' <VARNAME> '<=' <Value2> 'THEN' <WSL equivalent of Statement(s)>

5.7.4 Arithmetic

The conversion from arithmetic expressions in COBOL to WSL is fairly straightforward with the following COBOL to (->) WSL equivalent statements:

'ADD' <VARNAME1> 'TO' <VARNAME2> 'GIVING' <VARNAME3> ::= <VARNAME3> ':=' <VARNAME1> '+' <VARNAME2>

'ADD' <VARNAME1> 'TO' <VARNAME2> ::= <VARNAME2> := <VARNAME1> + <VARNAME2>

'SUBTRACT' <Operand1>, <Operand2>, <Operand3> 'FROM' <VARNAME1> ['GIVING' <VARNAME2>] ::= <VARNAME2> := <VARNAME1> - <Operand1> - <Operand2> - <Operand3>

'DIVIDE' <VARNAME1> 'BY' <VARNAME2> 'GIVING' <VARNAME3> 'REMAINDER' <VARNAME4> 'ON' 'SIZE' ERROR' <Statement> ::= <VARNAME3> := <VARNAME1> '/' <VARNAME2> ';' <VARNAME4> := <VARNAME1> 'MOD' <VARNAME2> ';' 'IF' 'SYSTEM.ERROR.ERRORTYPE' '=' 'On Size Error' 'THEN' <WSL equivalent of Error-Handling Statement> 'FI'

Mod is a predefined WSL function that gives the remainder value after division.

5.7.5 Error and Interrupt Handling

Because of the difficulty in modelling interrupts and exception handling in WSL, all error-handling and exception-handling will be done in-line (Younger, 1993). Inline error-handling determines if an error has occurred immediately after the codeline which may have created the error.

This COBOL-WSL converter uses a pre-defined WSL construct, System, in order to model system interrupts and error handling. The System construct has several fields which, in turn, have several sub-fields. The System fields include:

DateTime – which returns the system time

Interrupt – this represents system-generated interrupts

Error – this represents system-generated errors

These fields include sub-fields as well. Interrupt has a subfield, Interrupt Type, which holds info such as interrupt type and another field, Interrupt_Status, which serves as a flag to indicate that an interrupt has or has not occurred. The field, Error, contains sub-fields, ErrorType, that holds info such as error type and another field, Error_Status. A polling control construct

occurs after each WSL operation statement, as in the example below. This construct polls the System.Error or System.Interrupt (implicit structs of the System object) to determine if an interrupt or an error has occurred. If one has occurred, the appropriate error/interrupt handler is called to manage this occurrence.

An example of inline error handling:

```
Proc GenericProc()
S := 4 / 0
If System.Error.ErrorType = 'DivisionByZero' then
  Call <Error-Handler-Name>
Fi
```

5.7.6 Peripheral Devices

COBOL-reserved words "PRINTER" and "KEYBOARD" are treated like file shunts in WSL, with their filenames "PRINTER" and "KEYBOARD" pre-declared/pre-defined in WSL. Keyboard has some pre-defined variables used for its indexing: Keyboard_Buffer_Start, Keyboard_Buffer_End, Keyboard_Buffer_Index.

Certain keywords are used to indicate a particular peripheral device associated with a physical shunt. The particular device associated with a particular keyword follows this format: Printer: *IOPrinter*, Keyboard:*IOKeyboard*, Terminal:*IOTerminal*. Because these peripheral devices are assumed to have no index – these peripheral devices access the first element of the list associated with that peripheral devices; peripheral devices that produce output always overwrite the first element of the list associated with that device with new output; peripheral devices that consume input always consume the first element of the list associated with that device.

Consequently, to accept characters from a keyboard or to display characters to a screen, the following COBOL to WSL rules are used respectively:

'ACCEPT' <VARNAME> **::=** 'Fetch' <VARNAME>, 'IOKEYBOARD', <CURRENT-KEYBOARD-INDEX>

'DISPLAY' <VARNAME> **::=** 'Put' 'IOTERMINAL', <VARNAME>, <CURRENT-TERMINAL-INDEX>

5.7.7 System Calls

A common example of a system call is the obtaining of the current system date and storing this value into a variable. The rule that provides the COBOL to WSL syntax is as follows:

'ACCEPT' <VARNAME> 'FROM' 'DATE' **::=** 'Fetch' <VARNAME>, 'System.DateTime'

DateTime within the System structure is a predefined WSL function returning the system time.

5.8 Summary

Because of the large number of COBOL dialects and the huge number of COBOL constructs, a number of technical issues arise when trying to parse COBOL code. These technical issues, in turn, make it very difficult to find feasible tools that would be able to parse the COBOL code of the selected legacy system to its WSL intermediate representation. These technical issues also make it very difficult to compile a comprehensive set of rules to convert a system written in even a single COBOL dialect to a WSL equivalent program. However, a set of common rules to construct a WSL equivalent program from the COBOL source code of this selected legacy system is provided, along with explanations of these rules.

One issue that arises during the COBOL to WSL conversion process is that, in batch-oriented systems much data manipulation is involved in the form of files. WSL, at present, does not have a defined set of constructs to handle such common COBOL file constructs as indexed files, file sorting routines, and index keys. The development of such constructs is outside the scope of this thesis and an area of future research.

Chapter 6: Class Diagrams and Object Identification

6.1 Introduction

The first step in restructuring a procedurally structured legacy system to an object-oriented system is to identify possible objects within the legacy source code and then restructure the procedures and variables of the old system into methods and attributes respectively of objects of the new system.

In this chapter, the development of the thesis' object identification method, along with its algorithm, is described along with its validation via the replacement lemma and via experimental results (in comparison with selected other methods of object clustering). Using this thesis' method of object clustering, a method of extraction of class diagrams with classes and their associations is described.

6.2 Development of Object Identification Method

In order to obtain the original high-level structures of a program from its legacy source code, object identification methods were developed on models of those of Gall, van Deursen, and Newcomb (Gall, 1998; Newcomb, 1998; van Deursen, 1999a). Newcomb and Kotik (Newcomb, 1995) use an object identification methodology which takes all level 01 COBOL records as a starting point for classes. They then continue to map similar records to single classes, and find sections of the COBOL program that can be associated as methods to these records. This methodology can be highly automated and produces an object-oriented program that strongly resembles the original COBOL sources. De Lucia also uses persistent data stores, such as records and files, as the centres of potential objects (De Lucia, 1997). Gall identifies potential objects from source code by measuring the degree of coupling between different entities such as procedures and variables. If two procedures have a high degree of coupling, or interaction among themselves, these two procedures probably should be assigned to the same object class. Gall also uses data dependencies between procedures and variables in order to identify potential objects. If two variables share a common data dependency, such as both of them belonging to the same record structure, these two variables should probably be assigned to the same object class (Gall1998). Van Deursen places procedures with high fan-in or fan-out into separate classes (van Deursen, 1999a).

Thus, given a sample WSL program, it was proposed to determine the effect that the level of atomicity chosen, whether record level or record field level, for data analysis has on class identification and the degree of coupling.

The first step of this proposal is to scan the program code. Variables that are used within WSL control structures, such as "if" or "while" statements, are classified as control variables. Variables that have been used in WSL data assignment statements are

classified as data variables. Control variables are further subdivided into control variables that are used as guards for WSL constructs which call other procedures and those guard that do not. This subdivision is used in a UML representation of this WSL program where the latter variables are used to represent guards for transitions in UML activity diagrams and the former are used to represent guards for intra-class procedural call events that invoke these procedures in UML sequence diagrams.

An example of this control variable acting as a guard for an event, there is the following sample WSL program segment:

PROC READ-FILE

IF FILE-EOF **THEN**

CALL ERROR-FILE-READ()

FI.

Assume that Proc READ-FILE is assigned to Class A while ERROR-FILE-READ is assigned to Class B. When the control variable, FILE-EOF, is true, this condition signals an event – an invocation of method ERROR-FILE-READ in Class B by Class A. By using this method, a strictly sequential, procedural-driven program is transformed into a pseudo-event-driven program.

Regardless of their classification as data or control variables, variables which are used within WSL procedures will be assigned to objects whose attributes or variables they access the most frequently within the procedure. If a variable is assigned to one class object and a procedure, assigned to a different class object, accesses that variable, an object class association is created between the two object classes. If these procedures access this variable, it is a one-to-one association. After these tasks have been completed, the WSL representation is then transformed from that of a strictly sequential, procedural program to an object-oriented one.

By using a similarity/dissimilarity matrix, variables that are used within WSL procedures may be assigned to objects whose attributes or variables they access the most frequently within the procedure. The exceptions to this assignment of variables/procedures to classes that they access the most frequently are if the procedures are determined to be either externally defined, have high fan-in, or have high fan-out. If a variable is assigned to one class object and a procedure, assigned to a different class object, accesses that variable, an object class association is created between the two object classes. Because all of the reengineering classes are single instance objects, it is a one-to-one association.

Using another of van Deursen's object identification techniques, it was determined which procedures have a high fan-out (procedures which call many other procedures) and which procedures have a high fan-in (procedures that are called by many other procedures). Procedures with a high fan-out are usually control modules and thus should be kept in a separate control class. High fan-in procedures should likewise be put in their own separate class. Although combining these error-logging functions within their most often-used class object may reduce the number of inter-object couplings, it has the undesirable effect of increasing coupling between the class containing these error-logging functions and other classes whose only commonality with the former class is through the logging functions. Hence, an obscuration of the real purpose of the function within the system design is created when the high fan-in and fan-out procedures are included in the data clustering analysis (van Deursen, 1999a).

In (Millham, 2002), several different methods to identify possible source objects from legacy system code based on static code analysis were examined and then the method that produces objects with the least amount of inter-object coupling was selected. All of these methods used the same sample of code. One method used Newcomb's method where records and their associated fields form distinct class objects. The second method used similarity/dissimilarity methods, both procedural and variable usage, at the record field level atomicity but without separation of high fan-in and high fan-out functions, along with their associated variables, into separate classes. The third method was similar to the second method except that it separated high fan-in and fan-out procedures, along with their associated variables, into their own classes and, consequently, this separation removed these procedures and variables from any further consideration in object clustering. The results of these methods were examined in regards to the number of objects that they produced, the degree of coupling among the classes, and the type of inter-class coupling.

After investigating the results of several samples of code and the classes, attributes, and methods identified from them, the following conclusions were reached. Comparing the class objects identified and the degree of coupling between these objects produced using an analysis of all procedures versus that produced using an analysis of those procedures without high fan-in or fan-out, the former approach seems to group the majority of procedures into one large object with the remaining procedures grouped into two much smaller objects. Although combining these error-logging functions within their most often-used class object may reduce the number of inter-object couplings, it has the undesirable effect of increasing coupling between the class containing these error-logging functions and other classes whose only commonality with the former class is through the logging functions. Hence, an obscuration of the real purpose of the function within the system design is created when the high fan-in and fan-out procedures are included in the data clustering analysis. All of the variables, whether control or data, which are shared between more than one procedure were part of a record structure. Had these record structures formed the nucleus of our classes with the record structure forming a closer coupling factor than actual usage of its record, the object identification would be quite different. Function driven couplings (procedure calls) between procedures remain the same however, shared variables are treated differently. Whereas in record field level atomicity, variables are assigned to the procedure which utilises them the most; in record level atomicity, records are assigned to the procedure which utilises them the most and any fields of these records are, therefore, assigned to this procedure regardless of their individual level of usage by that particular procedure. As a result, the class structure looks quite different from that if the level of atomicity was at the record field level; furthermore, the degree of coupling is more evenly spread among the objects (50% among all three classes as opposed to 100% between two classes for record field level atomicity). From our variable usage analysis, we determined that very few variables, 6 out of 135 or approximately 4%, are shared among procedures. However, half of these shared variables act as control variables. These control variables are used to signal a particular state or invoke an event. Of these control variables, a majority (approximately 66%) was used to invoke a method outside of the attribute's owning class. Consequently, it seems that few variables are shared among the procedures. Half of those variables that are shared are used to signal a particular state or event among procedures. Thus, by utilising these control variables as signals for a particular event occurrence, this procedural-driven WSL program can be transformed into an event-driven program (Millham, 2002).

As a result of this experiment, it was decided that the third method, using similarity/dissimilarity matrices of usage among procedures and variables along with the separation of high fan-in and fan-out procedures into separate classes, would be the optimal object clustering method. This method was further enhanced by the separation of external procedures into their own class. External procedures are procedures, which may be called in one program file, but are defined elsewhere, either in another program file or within another system itself. By separating external procedures into their own classes, a separation is made between procedures that are defined within a program file and belong to this file by some sort of programmer-defined logic and

external procedures, which should belong to the system or file where they were defined. In addition, the exclusion of externally declared procedures prevents the occurrence of the same procedure being placed in two or more different classes (Millham, 2002).

The thesis approach in object identification is use individual record fields, not records, as entities of evaluation for classes, unlike Newcomb's method. Gall's degree of coupling algorithm is used to determine which variables and procedures are most closely coupled to specific classes. Van Deursen's algorithm that places controller and logging functions into separate classes eliminates classes whose attributes/methods include those with only functional cohesion. By identifying objects and any associations, whether procedural calls or shared variables, UML class diagrams, with inter-class associations, can be derived. The thesis' object clustering technique is based on a combination of methods of Gall, Newcomb, and van Deursen, along with the separation of externally defined procedures into separate classes.

6.3 Extensions to WSL for Object Orientation

Once the objects are identified, it is necessary to restructure the procedural representation into an object-oriented one. Procedures and their related variables are encapsulated within a new structure, a class. Consequently, there was a need to define new structures, namely a Struct structure, which is used to hold related variables within a record structure, and a Class structure, which is used to hold related procedures and variables within a class structure.

After the encapsulation of procedures and variables within class structures has completed, the parameter list of each encapsulated procedure is modified to reflect this class structure. If a method accesses an attribute of its own class, this attribute does not need to be passed in as a parameter to the method. Instead, this attribute can be referenced directly within this method. Whereas in procedural WSL, all variables within a procedure must first be passed in as a parameter; in object-oriented WSL, only variables that are used within a procedure but belong to a different class than that of this procedure need to be passed in as a parameter.

The **Var** construct is extended to include structures of both record and *Class* type. A record structure is defined by using the Var Struct keywords in the following format:

```
Var Struct RecordName
Begin   /* beginning of record fields */
    Var VarName1
    Var VarName2
End  /* end of record fields */
```

The class structure is defined by using the *Class* keyword in the following format:

```
Class ClassName
Begin
    Var VarName1  /* Attributes of Class */
    Var VarName2
```

Proc ProcName2 /* Methods of Class */

Begin

/* WSL statements */

End.

End

To access a record field enclosed within a record data structure, the following format is used:

RecordName.FieldName

To access a class attribute or method enclosed within a class structure, the following format is used:

ClassName.AttributeName

6.4 Object Identification Algorithm

Outlined below is the algorithm used to parse and statically analyse the WSL source code to determine the degree of coupling between the code's various procedures and variables and partially based on this coupling, cluster closely-coupled variables and procedures into their relevant classes.

1. Parse WSL source code.
2. Determine current procedure as indicated by the Proc <ProcName> declaration structure. <ProcName> becomes the current procedure name until the "end." term is discovered, which indicates the end of the procedure construct and its associated code.
3. When the Proc <ProcName> declaration is encountered, this declaration is followed by a list of parameter variables. Each of these parameter variables is added as a variable to the ProcInVarList matrix. The ProcInVarList is a three-dimensional matrix consisting of the following dimensions: current procedure name, variable name that is used within the procedure, and usage of that variable. This usage dimension is initially zero but each time a variable V is encountered/used within procedure P, the usage column within the row of ProcInVarList P V is incremented by one.
4. If a procedure is called by another procedure, which is denoted by the term CALL <ProcName>, <ProcName> becomes the called proc P1 and the current procedure in which this term was encountered, becomes the caller proc P2. Similarly to the Variable usage, a three-dimensional matrix is used with the dimensions: CallerProc, CalledProc, and Usage. Usage is initially zero but whenever a CALL <ProcName P1> term is encountered in procedure P2, the usage count column of row CalledProcList P2 P1 is incremented by one.
5. Sort the ProcVarList matrices by Usage descending. Consequently, after sorting, in the ProcVarList matrix, the variable is assigned to the procedure where it has its highest usage. In order to avoid assignment conflicts, before a variable is assigned to a procedure, a check is made to ensure that the same variable had not previously been assigned to a different procedure.

6. Find procedures that are called by other procedures but call no procedures themselves (logging procedures) or procedures that call other procedures but call no procedures themselves (controller procedures). The maximum usage of procedures that are called by other procedures but call no procedures themselves are found; all procedures that are called by other procedures but call no procedures have their usage evaluated against this maximum usage. If the usage is within a certain range (in the 80th percentile), the procedures are considered as logging procedures. Similarly, the maximum usage of procedures that call other procedures but are not called by other procedures are found; all procedures that call other procedures but are called by no other procedures have their usage evaluated against this maximum usage. If the usage is within a certain range (in the 80th percentile), the procedures are considered as controller procedures. These procedures are placed in separate Logging and Controller classes respectively and these procedures are removed from future consideration in terms of class aggregation.
7. Find procedures (externally-defined procedures) that are called within other procedures but are not defined within the same program file. Place these procedures into separate classes to denote their external definition and these procedures are removed from future consideration in terms of class aggregation.
8. Similarly, the CalledProcList matrix is sorted by Usage descending, grouping by Current Procedure name. The called procedure, CalledProcName, is assigned to the caller procedure, CallerProcName, which calls this procedure the most. If a procedure is neither in the Controller or Logging class and is not assigned to another procedure using the previous step, this procedure is put in the UnassignedProcedure list.
9. Each group of assigned procedures forms a class. Each procedure in the UnassignedProcedure list becomes a separate class.

6.5 Extraction of Class Diagrams and Associations

Class diagrams represent the static structure of the system. Class diagrams convey information about classes used in the system such as their properties, their interfaces, and how these classes interact with one another (Gall, 1998). Associations are relationships between instances of two classes.

After this reverse engineering process transforms the legacy system from its procedural structure to that of an object-oriented one, TAGDUR extracts the class diagram from the transformed system. Class definitions from the system are modeled as classes in the UML class diagram; variables encapsulated within a class become class attributes and procedures associated with a class become methods in the UML class diagram.

Classes in this system are grouped into UML packages. A package, in this legacy system example, corresponds to the original COBOL copybook. The assumption is that the original programmers divided up the system into modules, in this case COBOL copybooks, according to some logical criteria. This logical modularisation of the system is preserved in the form of packages in UML diagrams. These packages with their classes can also be considered frameworks of classes with each framework retaining the logical partitioning criteria that divided the original system.

In this sample legacy system, developer comments within the code are confined to comments at the top level of the software source code file. These comments typically give information such as the author of the program, maintenance history, and other purposes. In order to parse and analyse these developer comments, these comments must be expressed in a consistent and clear manner. Given the wide variety of developers that developed and modified this program over the years and the wide variety in the way that people may express the same concepts, it is difficult to extract the meaning of these comments from the basis of natural language. As a result, these comments are simply put in the UML element of Comment for perusal and comprehension by developers. Compounding the difficulty is that with natural language comments, it is difficult to determine what sections of code that these comments apply to. An example, do these comments apply to the whole source code file, a specific program, an individual codeline (and if so, which codeline)? Given the wide variety that the placement of comments relevant to the section of code are recommended, determining which UML diagrammatic element, such as a class or method, should be attached via a comment is very difficult. Consequently, all comments are attached to the UML Package element.

Classes that have been identified within legacy system may be aggregated into a super-class hierarchy using a number of criteria. Highly-coupled groups of classes or classes having a compositional relationship may be grouped into a super-class hierarchical structure. Since this system makes much use of logical records, logical files, and physical files, these entities, if placed in separate classes, can be depicted in a class hierarchy. Logical Record Classes, R1 and R2, have a 1:1 compositional relationship with their logical file name classes, L1 and L2. Because many logical record files may access the same physical file, the logical file classes have a 1:many compositional relationship with their physical file object. Each of the physical files, in turn, have a 1:1 compositional relationship with the pre-defined object, File, which is the super-class of all file I/O classes.

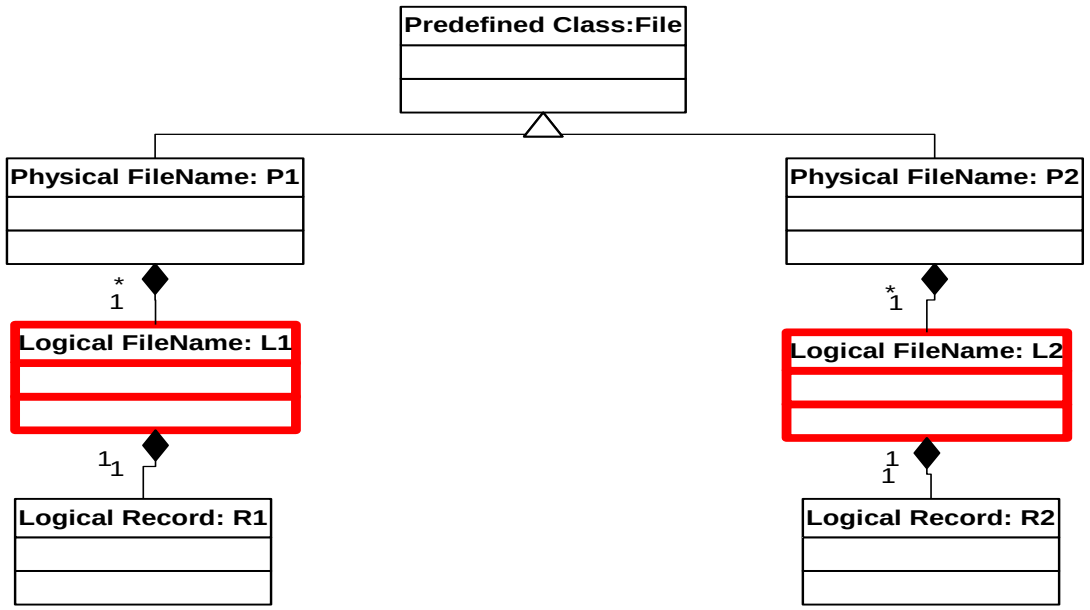


Fig.6.5.1 Class Diagrams with their Associations with Various Logical and Physical File Classes Hierarchy.

The thesis tool, TAGDUR, models accesses between classes of other's attributes or methods as static associations between classes. Each end of an association contains a multiplicity; the multiplicity of an association end is the number of possible instances of the class associated with a single instance of the other end. Depending on the ratio of intances of a class accessing the attributes/method to the instances of classes accessed, the multiplicity of these association ends may be modeled as many-to-one, one-to-one, etc. (Ward, 1992). An example, if many instances of Class A access multiple methods and attributes of a single

instance of Class B, this association end would be modeled as many-to-one. If an instance of Class A access only one attribute of a single instance of Class B, then this association end would be modeled as one-to-one multiplicity.

The information gained during the object identification, regarding inter-class variables and procedures, transformation process of this system is used in modelling these multiplicities. During the object identification process, TAGDUR constructs two matrices: one matrix is a procedural usage grid which records the number of times procedure A is called by procedure B and the other matrix is a variable usage grid which records the number of times variable A is accessed within procedure B. These matrixes are used during the object identification process where highly coupled procedures and variables are grouped into classes. These matrices are also used when modeling UML diagrams. Variables or procedures that form attributes/operations of one class, class A, but are accessed by procedures that form operations of another class, class B, are modelled as an association between classes A and B. These two usage matrices are used when modeled the multiplicity of these associations. Using both procedural and variable usage matrices, the number of times that different variables and procedures of object A are accessed by procedures of object B form the type of multiplicity relationship between classes A and B. An example, if the procedures of multiple objects of class B access different variables and procedures of object A ten times, the association between object A and objects of B is modeled as an association of 1:n multiplicity.

6.6 Correctness Validation of Object Oriented Restructuring Algorithms

In order to provide some validation of the restructuring of this procedural, legacy system to an object-oriented system, the following proof is provided. The thesis' restructuring involves affixing a class prefix to a class attribute, formerly a variable, or class method, formerly a procedure, in order to access this class member. Previously, the variable or method would be accessed by a reference using their name without the class prefix. By encapsulating the variable or procedure within a structure, such as a class, it is necessary to reference the structure, along with the encapsulated member, when accessing this member. The following proof provides a validation of the renaming variables and procedures by affixing a class prefix.

According to classic object oriented theory, instances of a class are separate from the class that they represent. The class represents the type and the instance of the variable in programming language terminology. Object oriented theory allows the use of information hiding of members through the use of public, protected, and private members of the class; the information hiding ability controls the degree of access to these members outside its constituent class (Lippman, 1998). However, WSL is typeless. Thus, in WSL each class is equivalent to a class instance. Each class has exactly one instance and each class is unique. Furthermore, TAGDUR does not have the ability yet to do any pattern matching, other than physical composition, of classes according to behaviour, structure, or business logic so that the ability for class aggregation with a hierarchy of super and sub classes is limited. Furthermore, WSL has only locally scoped variables rather than varying degrees of access – thus, implementing a degree of access via private, protected, and public members is infeasible.

Object oriented theory requires that a prefix be attached to the name of the class member, whether an attribute or method, that is being accessed. This prefix indicates the member's constituent class instance. If the member is accessed outside its constituent

class, this class instance prefix is unneeded or may be designated with the self prefix. In C++, this self prefix is the "self" keyword (Lippman, 1998).

In order to maintain consistency, member prefixes in WSL are always the full name of class instances, regardless of whether these members are accessed outside their class or not. Thus, attribute A1 of Class C1 is accessed as C1.A1 regardless of whether this access occurs in a method of class C1 or not.

According to Lemmon, the replacement theory states that

> "If A is equivalent to B, and D results from C by replacing some occurrence of A in C by B, then C is equivalent to D, for any formulae A,B,C, D." (Lemmon, 1988: p 193).

Thus, according to the replacement theory, the renaming of variable V1 to V2, if replacement is consistent, where V1 is equivalent to V2, enables V2 to replace V1. Object oriented theory states that when a variable (or attribute) V1 is enclosed within a class structure, the same variable V1 must be accessed, from without its own enclosing class, by the class prefix, C1. Similarly, when procedure (or method) P1 is enclosed within a class structure, the same procedure must be accessed, from without its own enclosing class, by the class prefix, C1. Thus, V1 becomes C1.V1 and P1 becomes C1.P1. Consequently, if all instances of V1 are consistently replaced by C1.V1, this replacement, by the replacement theory, enables C1.V1 to be equivalent to V1. Consequently, by employing the replacement theory, object oriented variables, enclosed within a class, are equivalent to their matching procedural ones as demonstrated by this proof.

6.7 Evaluation of Proposed Method and Algorithms

The thesis' method is an enhancement of Newcomb's, Gall's, and van Deursen's methods for identifying clusters of related data and procedures. Newcomb and van Deursen use existing COBOL records as a starting point for object identification. Hence, all record fields of an existing COBOL record become the attributes of a new class, as formed by the COBOL record. Newcomb measures the coupling between existing COBOL records in order to form associations among classes. Van Deursen separates, from data clustering, those procedures that have an high fan-in or fan-out which often denote a functional (logging/controller) coupling only with other procedures. Gall uses static analysis of the control and data flow within the code in order to group the most frequently accessed variables and procedures into the same class. A new enhancement is to exclude externally defined variables from object clustering (Newcomb, 1999; van Deursen, 1999a).

The thesis' method measures the coupling of variables, at the individual record field level rather than the record level. This finer level of granularity in measuring coupling is one advantage. COBOL typically has a large number of record fields within its records. Over the long maintenance history of the typical COBOL legacy system, new record fields typically are appended to existing records rather than put in a new, more logical record that is related to existing records. As a result, if records are used as a starting point for classes in Newcomb's method, all record fields within that record form attributes of the new class. Due to the maintenance practice of appending new record fields to existing rather than, more logically, new records, not all record fields are logically related to the record structure that they are assigned to. Without this logical relation, these record fields do not have optimal cohesion with their record structure (van Deursen, 1999a). By considering the coupling of each record field rather than its entire record, the record field will be assigned, as an attribute, to the class with which it forms an optimal coupling. Experimental

results (Millham, 2002) indicate that this finer level of granularity that the specified method allows the determination of classes with less inter-class coupling between variables and procedures.

Another enhancement of the specified method, which incorporates part of van Deursen's object identification method, is to eliminate procedures with very high fan-in or fan-out from object identification. Instead, these procedures are put in classes of their own (Logging and Controller classes respectively). Typically, these procedures perform a logging or controller function with only functional cohesion with other procedures. The thesis' experiments (Millham, 2002) demonstrate that if these procedures are not put in separate classes but are considered in the specified object identification algorithm, the classes produced are much different. The advantage with this thesis method is demonstrated as the number of classes produced is smaller with a larger group of procedures; furthermore, the coupling between these procedures and their related variables within the same class is much less than if these high fan-in and high fan-out procedures are put in separate classes.

The chief disadvantage of this method is that it is based on a static analysis of code rather than a dynamic analysis. This static analysis measures the number of calls made in code to each procedure by a given procedure (static analysis) but does not measure the number of calls made to each procedure by a given procedure during a typical run given a typical set of test data. Consider, for example, the following fragment of code:

```
Proc A(X)
 Begin
Call B
While X> 3 do
 Call C
 X := X -1
od
End.
```

A static analysis of the code would reveal procedure A calls procedure B and C once. However, a dynamic analysis would reveal procedure A calls procedure B once, and calls procedure C zero or more times depending on the value of C. With a set of typical input data, a dynamic analysis would reveal the average number of times procedure C is called from procedure A. A dynamic analysis would give a more accurate picture of the degree of coupling (such as the number of times a procedure is called by another) between each procedure and, as a result, procedures are incorporated into the classes with which they share multiple classes. One major problem of a dynamic analysis is that it is dependent, on the set of "typical" input data. Is this set of input data truly representative of the data inputted into the system? Also, the "typical" set of input data may change as the business environment that the legacy system models changes. Because of the difficulties in obtaining a truly representative set of input data, it was decided to base this reengineering effort on static analysis of source code only rather than basing it on a dynamic analysis of the system in relation to a given set of input data (Bertuli, 2005; Egyed, 2003).

Identifying objects via static, rather than dynamic, code analysis was selected for many reasons. A dynamic code analysis requires a careful trace, given a typical set of input, of the coupling between various variables and procedures in the code during several test runs. Identifying a typical set of input for a system is difficult because the typical input handled by a system often changes

over time and may vary greatly depending on such factors as the particular time period chosen (Egyed, 2003). Furthermore, because of data confidentiality rules, data representing a typical set of inputs for this particular system was unavailable.

An additional advantage in considering variables at the record field rather than the record granularity is that it separates record fields from their existing record structure. This approach is useful because the record structure may change during later data reengineering at a later stage and basing classes on existing record structures would be counter-productive.

6.8 Summary

The thesis' object identification process is necessary not only to cluster related variables and procedures into classes but also by using the same variable or procedural usage matrices that were used in this process, associations between classes can be identified. Furthermore, these matrices may be used further in determining the navigability of these associations. This information is used directly in modelling UML class diagrams.

However, the specified object identification algorithm identifies object according to variable and class coupling but not necessarily cohesion. Cohesion can be defined as the level of uniformity in terms of the system's design goal. Without information regarding the problem domain of the selected legacy system and relying on source code alone, it is difficult to determine what procedures and what variables are cohesively related in regards to the system's overall design. Thus, developing loosely coupled classes with high cohesion is difficult in reengineering.

This object identification algorithm identifies clusters, which are deemed classes. It could be argued that these classes are really objects since in the selected restructured system, each class consists of one instance or one object. This restructured system is not really a true object-oriented system; a true object-oriented system would consist of multiple instances of the same class within a system. A true object-oriented system is often feasible during forward engineering when the real world environment upon which the system models is well-known and a mapping of real world objects to system objects can be accomplished. With the knowledge of this real-world environment, a hierarchy of classes can be constructed. An example, a Person super-class can be defined with a sub-class of Customer. A bank may have multiple instances of class Customer. However, this class hierarchy along with the relationship of objects to their classes is very difficult to accomplish when this real world environment is unknown, along with the relationship of the real world objects and their possible system objects, and the only information on which to base the specified object identification methods is the source code itself.

However, the specified object identification algorithm which produces a one class per object correspondence is practical, give the above constraints, in restructuring COBOL into objects from which UML diagrams can be extracted. UML is class dependent for its modelling, not only in its class diagrams, which model classes as its structure, but in other diagrams, such as sequence diagrams, that model the interaction among classes.

An investigation was conducted, from among several methods, in order to determine which object clustering method produced objects with the most cohesion and least amount of inter-class coupling. The results from Newcomb's method of using record structures as class seeds were compared to Gall's method, which relied on a similarity/dissimilarity matrix, in order to cluster data

and methods that most commonly used each other into classes. A third method of van Deursen's was investigated: Gall's reliance on similarity/dissimilarity matrices combined with van Deursen's pattern matching to identify high fan in and fan out procedures and the separation of these procedures into separate classes. It was discovered that this third method produced objects with the most cohesion and the least amount of inter-object coupling because it separated high fan in and out procedures, which often play a logging and controller role respectively and which have only functional cohesion with their coupled objects, into their own separate classes. Consequently, this third method was adopted. This third method involved analysing individual variables and record fields, rather than simply records, along with procedural call graphs, according to their usage. Van Deursen's pattern matching was used to discover procedures with high fan-in and fan-out which were then placed in separate classes and remove from future consideration in object clustering. Externally defined procedures were also placed in separate classes and removed from future clustering consideration. As a result of the adoption of this method, objects with high cohesion and lower inter-object coupling were identified and procedures, such as controller functions, with only functional cohesion to the identified objects, and externally declared procedures, with no logical purpose with the identified objects, were placed in their own separate object classes.

Chapter 7: Sequence Diagrams and Independent Task Identification

7.1 Introduction

In this chapter, several algorithms are outlined that determine the execution of tasks at various levels of granularity: individual codeline, procedural, and program block. A validation of these algorithms is provided through carefully selected test cases contained within a test program. Each of these independent task evaluation algorithms plays a role in the correct modeling of UML diagrams from source code. Independently executing codelines are modeled as parallel flows in activity diagrams and as concurrent events in sequence diagrams. Event identification involves parsing the source code for pseudo-events such as inter-object procedural calls, file I/O, and calls to peripheral devices. Other event identification processes involve modeling events such as error handling and system interrupts. Once these events have been identified and the task encompassing this event has its task independence determined, the sequence number of events modeled in the UML sequence diagrams can be correctly depicted and this UML diagram can be extracted.

7.2 Development of Independent Task Identification Method and Algorithms

After the legacy source code has been converted into a WSL representation and this WSL representation has been analysed for potential class object clusters, this WSL representation must, in turn, be analysed in order to determine independent and non-independent tasks.

This investigation involves determining what level of granularity of tasks (procedural, programming block, or individual code line) is the best for determining task independence. This approach consists of four steps:

1. Determining Data and Control Dependencies – the specified method of determining whether two tasks T1 and T2 are independent involves identifying whether any control or data dependencies exist between these two tasks. A dependency can be defined where one unit of execution, or task, T1 is dependent on another task T2 for its execution. These dependencies may be a control dependency where the expressions, and their constituent variables, of the control constructs enclosing the task determine if and how this task will be executed. The dependencies may also be a data dependency where two tasks, T1 and T2, share common variables and where the correct computation of results in task T2 is dependent on the preceding computation of results using shared variables in task T1.

2. Program Block Identification – A program block is a logical unit of execution and it represents an intermediate level of possible explicit granularity in this legacy system. A procedure will contain one or more programming blocks and a programming block contains one or more code lines.
3. Individual Codeline Evaluation – a code line, or statement of legacy code, represents the minimal granular unit possible in this legacy system. An individual code line represents the lower bound of possible explicit granularity in this legacy system.
4. Procedure Granularity - a procedure, in a procedurally-designed system, is one of the largest self-contained granular units possible. According to procedural design theory which was prominent in programming during the time that this legacy system was developed, a procedure should encapsulate one or more related functions along with their associated data, in the form of locally-used variables. By assessing tasks at the procedure granularity level, the granularity present in the explicit modularisation of procedural design can be taken advantage of. A procedure represents the upper bound of possible explicit granularity in this legacy system.

The detailed techniques can be seen in the remaining subsections.

7.2.1 Determining Data and Control Dependencies

While analysing the WSL code, all levels of granularity, in their determination of task independence, use a common function Check_For_Shared_Vars which has two parameters, one parameter is one set of codeline(s) and the other parameter is the immediately successive set of codeline(s). If one set of codeline(s) updates one or more variables that are currently being used with the other set of codeline(s), then the function returns true; otherwise, the function returns false.

7.2.2 Program Block Identification

For program block analysis, it is important first to identify and categorise the program blocks and then determine the task independence of each program block. This determination is accomplished by using the following algorithm:

Within each procedure, programming blocks must be determined. Programming blocks are delimited by keywords of the if-then or while-do constructs. In the case of nested programming blocks (1...n), each programming block is assigned a nesting level with 1 begin the outermost programming block and n being the innermost programming block. In order to distinguish between different programming blocks of the same procedure at the same level, each programming block of the same level is assigned a sequence of 1...n where 1 is the first programming block encountered and n is the last programming block encountered. In the case of procedures with no control logic, all lines of code within the procedure are grouped into a single programming block.

After all the programming blocks have been identified, determine, for each procedure, which programming block level is the most numerous. This most numerous programming block level determines, in the case of nested programming blocks, how each code line will be evaluated. All code lines belonging to a programming block of this most numerous level are evaluated as part of

this programming block, regardless if they belong to other programming blocks as well. For code lines that fall outside this particular programming block level, these code lines are assigned their most innermost programming block.

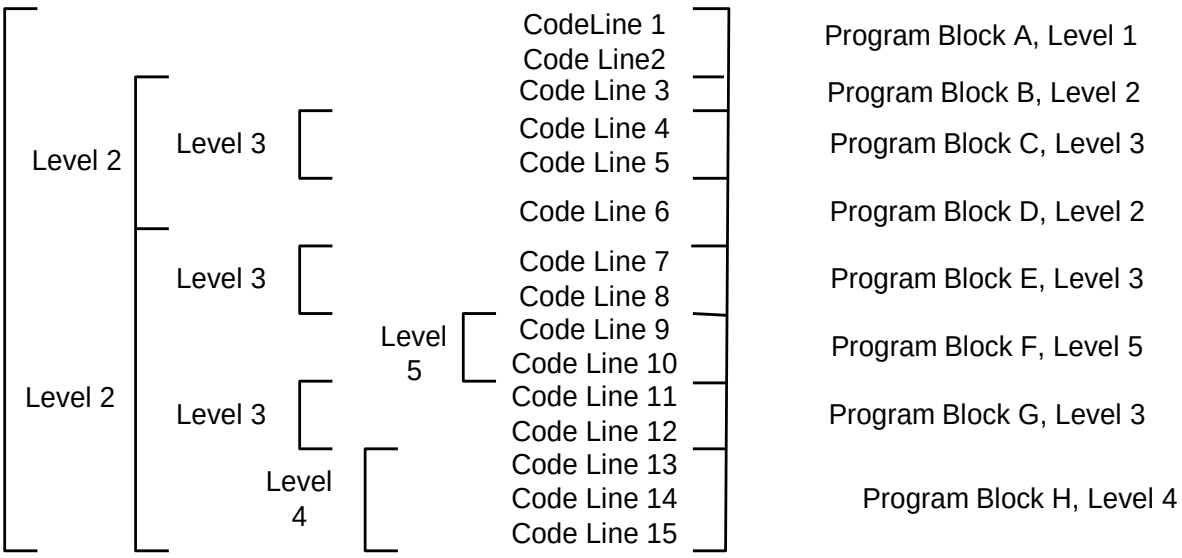


Fig. 7.2.2.1 Code Lines and their Levels.

The algorithm to determine which programming blocks are independent is as follows:

For each procedure, initialise the task sequence number to zero;

For each pair of program blocks within current procedure:

if Check_For_Shared_Vars(codeline(s) of first program block, code line(s) of second program block) returns true, then update code line(s) of first program block to the task sequence number; increment to next task sequence number; and update code line(s) of next program block to the task sequence number,

Otherwise {no shared variables} update code line(s) of first and second program block to same task sequence number since they both can execute in parallel.

Each pair of program blocks are managed so that the second program block of the current pair becomes the first program block of the next pair.

7.2.3 Individual Code Line Evaluation

The algorithm to check individual programming codelines is as follows:

For every procedure

Current task sequence number is initialised to 0;

For every code line (curcodeline) of current procedure

If curcodeline or next code line is a While or if construct, then update curcodeline's task sequence number to current task sequence number; increment to the next current task sequence number; update next code line's task sequence number to current task sequence number;

Otherwise,

If Check_For_Shared_Variables (curcodeline, next code line) returns true, then these two tasks are not independent tasks so update next code line to next task sequence number;

Otherwise, {no shared variables so these two tasks are independent};

Update curcodeline's task sequence number to current task sequence number;

Update next code line's task sequence number to current task sequence number.

Each code line is evaluated in context of its procedure rather than in context of its program. The reason for this evaluation is that procedures are a logical entity to encapsulate code lines and using this method, each code line is evaluated once, within the context of its procedure. If the code line was evaluated on a program-level basis and in relation to the procedural call graph, a code line may be evaluated several times.

If a control construct is encountered in a code line, the task sequence number is incremented to the next task sequence number. The reason for this incrementation is that any subsequent code lines that are enclosed by this control construct can not be evaluated in parallel with the control construct code line. The control construct code line must be evaluated first and then any enclosed code lines can be evaluated.

The maximum number of program blocks of the same level is selected in order to maximise the number of granular units that can be considered for parallelisation within a procedure.

7.2.4 Procedure Granularity

Independent task evaluation, at the procedural level of granularity, is necessary to determine the sequence of messages, which are inter-class procedural calls, among classes in the UML sequence diagram.

The algorithm to determine task independence among procedures is as follows:

Determine the main controller function(s), through analysis of fan-in and fan-out of procedures. Controller functions typically have a high fan-out (they call many other procedures) but have a very low fan-in (they are called by very few other procedures). In this case, the COBOL->WSL rules stipulate that the main controller function for a COBOL source file, whether COBOL program file or copybook, is created with the procedure name <SOURCE-FILENAME>_VAR-INIT;

The main controller function is given a procedure sequence number of 1 and assigned to a temporary table, TmpProcsAlreadyAssigned;

Increment the procedure sequence number variable, ProcSeqNo, by one to two;

All of the procedures that are called by the main controller function are then put into a temporary table, TmpDetermineParallelProcs, and then analysed for shared variables;

While updating_procedure_sequence_number is true

if one or more procedures, procsA, from the TmpDetermineParallelProcs share no variables with any other procedure in the TmpDetermineParallelProcs

set procedures', procsA, sequence number to ProcSeqNo

Assign procedures, ProcsA, to table TmpProcsAlreadyAssigned

Assign all procedures that are called from procedures, ProcsA, to table TmpDetermineParallelProcs

Delete procedures, procA, from table TmpDetermineParallelProcs now that their procedure sequence number has been assigned

Procedure_sequence_number is incremented by one

Set updating_procedure_sequence_number to true

Otherwise, set updating_procedure_sequence_number to false;

The While Loop continues until one of the three conditions is met:

a) The number of procedures to be analysed for parallelism is too small to be analysed – one remaining procedure

b) No updating of procedure task sequence numbers has been done and no procedures have been deleted from table, TmpDetermineParallelProcs;

c) The remaining procedures that have not been assigned a procedure sequence number now have this number assigned as a sequence number, incremented by one for each procedure. An example, the first remaining procedure is assigned for its procedure sequence number the procedure_sequence_number + 1, the second remaining procedure is assigned procedure_sequence_number + 2.

7.2.5 Procedural Level Independent Task Evaluation Algorithm

The algorithm to determine task independence among procedures is as follows:

1. Determine the main controller function(s), through analysis of fan-in and fan-out of procedures. Controller functions typically have a high fan-out (they call many other procedures) but have a very low fan-in (they are called by very few other procedures). In this case, the COBOL->WSL rules stipulate that the main controller function for a COBOL source file, whether COBOL program file or copybook, is created with the procedure name <SOURCE-FILENAME>_VAR-INIT;
2. The main controller function is given a procedure sequence number of 1 and assigned to a temporary table, TmpProcsAlreadyAssigned;
3. Increment the procedure sequence number variable, ProcSeqNo, by one to two;
4. All of the procedures that are called by the main controller function are then put into a temporary table, TmpDetermineParallelProcs, and then analysed for shared variables;
5. While updating_procedure_sequence_number is true

if one or more procedures, procsA, from the TmpDetermineParallelProcs shares no variables with any other procedure in the TmpDetermineParallelProcs

a)set procedures', ProcsA, sequence number to ProcSeqNo

b)assign procedures, ProcsA, to table TmpProcsAlreadyAssigned

c)assign all procedures that are called from procedures, ProcsA, to table TmpDetermineParallelProcs

d)delete procedures, ProcA, from table TmpDetermineParallelProcs now that their procedure sequence number has been assigned

e)procedure_sequence_number is incremented by one

f)set updating_procedure_sequence_number to true

otherwise, set updating_procedure_sequence_number to false;

2.The While Loop continues until one of the two conditions are met

a.The number of procedures to be analysed for parallelism is too small to be analysed – one remaining procedure

b.No updating of procedure task sequence numbers have been done and no procedures have been deleted from table, TmpDetermineParallelProcs;

3.The remaining procedures that have not been assigned a procedure sequence number now have this number assigned as a sequence number, incremented by one for each procedure. An example, the first remaining procedure is assigned for its procedure sequence number the procedure_sequence_number + 1, the second remaining procedure is assigned procedure_sequence_number + 2.

7.3 Correctness Validation of Independent Task Restructuring Algorithms

In order to prove the correctness of the specified algorithms to determine task independence, at the procedural and individual codeline level, it is necessary to validate that the specified algorithms are correct. This validation is through functional equivalence. A small sample program, outlined below, is taken and its values are calculated correctly in a sequential manner. This sample program contains many constructs typically found in legacy systems such as assignment, control (do and if), file I/O, and procedural calls.

The test program for validation was carefully selected to test many of the test cases, both common and boundary test cases, that the tool might encounter with a system to be reengineered. Procedures with shared variables (by reference) as well as independent procedures with no common shared variables were selected. In order to determine if the tool detected common dependencies, sections of code with shared data and sections of code with no shared data were used in order to determine if the tool detected these dependencies and accurately determined if these sections could be executed sequentially or not. Similarly, code encased within control constructs was given as an example in order to determine if the tool determined correctly if the variables within a

control construct would be considered as a dependency of variables used within that control block. Other more tricky examples were used, the same variable was used in a sequence of two contradictory computations based on sequence for determinism. File I/O examples were used to indicate if the tool determined that in the case of a variable being assigned a value, written to a file, had its value within the program changed, and then had its newly-altered value written back to the old value from the file, if these dependencies, file and program, would be correctly ascertained.

We evaluate this sample program using the specified independent task evaluation algorithm. Tasks that are deemed to be independent are executed either independently or have the original order of their execution modified. An example, the specified algorithm deems that Procedures A and B can execute independently and, in the original sequence ordering, Procedure A is executed before procedure B. The correct functioning of the specified task independence algorithm is determined, through functional equivalence, by reversing the order of Procedures A and B and then after execution, the final, and intermediate, results are compared to the original results produced during a strictly sequential execution. If both sets of results are the same, then it can be determined that the specified independent task evaluation algorithm at the procedural granular level is correct. In a similar manner, the independent task evaluation algorithm, at the codeline level of granularity, is evaluated using this functional equivalence testing.

100 lines of WSL code with control constructs (While and if), add, sub, I/O, procedures:

```
PROC WSLCODELINESFORINDEPENDENTTASKS_VAR-INIT(A, A2, A1, B, B1, B2, B4, C,
C_Top, C_Init, D, E, X1, F, F1, F2, G1, G2, G3, G4, H1, H2, H3, H4, H5, H6, H7, H8, H9, H10, H11,
H12, H13, I1, I2, I3, I4, I5, I6, FileA, FileB, J1, J2, J3, J4, J5, K1, FileC)
 BEGIN
/* main calling procedure */
 CALL Proc1(X1, A, A2, A1, B, B1, B2, B4, C, C_Top, C_Init, D, E)
 CALL Proc2(H13, H4, F, F1, F2, G1, G2, G3, G4, H1, H2, H3, H5, H6, H7, H8, H9, H10, H11, H12,
I6, I1, I2, I3, I4, I5, FileA, FileB)
 /* Proc 2 and 3 are inter-dependent via their common vars */
 /* CALL Proc3(H13, H4, I6, I1, I2, I3, I4, I5, FileA, FileB ) */
 CALL Proc4(J1, J2, J3, J4, J5 )
 /* independent proc */
 CALL Proc5(I6, K1, FileA, FileC)
end.

PROC PROC1(X1, A, A2, A1, B, B1, B2, B4, C, C_Top, C_Init, D, E)
 BEGIN
 A  := 1
```

```
 A2 := 4
 A := A1 + A2
 /* A = 7 : Point 1 */
 B := 4
 B1 := 6
 B2 := 3
 B4 := 8
 B := B1 + B2 * B4
/* B := 30 : Point 2  */
 C := 4
 C_Top := 8
 C_Init := 4 + 5
 while (C < C_Top)
  D := C_Init * C
  C := C + 2
 od
/* 2 iterations: D = 54 : Point 3  */
 if (D > 50)
/* should be true */
  E := D - C_Init
/* E := 45 : Point 4  */
  X1 := 36
 else
  X1 : = 12
 fi
 /* end of proc 1 */
end.

PROC PROC2(H13, H4, F, F1, F2, G1, G2, G3, G4, H1, H2, H3, H5, H6, H7, H8, H9, H10, H11,
H12)
 BEGIN
 F := 3
```

```
 F1 := F + 4
 F2 := F1 * F
/* F2 = 21 : Point 5  */
 While F1 > 0 do
  F := (F + 1) * 3
  F1 := F1 -1
/* after 7 iterations, F = 9840 : Point 6  */
 od
 if F > 1000 then
   G1 := 9
 else
   G1 := 12
 fi
 G2 := G1 * 3
/* G2 = 27 : Point 7 */
 G3 := G2 + 3
/* G3 = 30 - an example of two contradictory computations based on sequence for determinism:
Point 8 */
 G4 := G3 + 6
 H1 := 3
 H2 := 4
 H3 := 6
 H4 := G3 + 8
/* H4 = 38 : Point 9 */
 H5 := G2 + 3
/* H5 = 30 : Point 10 */
 H6 := 9 * 1
 H7 := 8 + 1
 H8 := (8 * 2) + 4
/* H8 = 20 : Point 11 */
 H9 := 12
 H10 := 3
```

```
 H11 := 4
 H12 := 3 + 3
/* H12 = 6 : Point 12 */
 H13 := H9 + H6 + H4
/* H13 = 59 : Point 13 */
CALL Proc3(H13, H4, I6, I1, I2, I3, I4, I5, FileA, FileB )
end. /* end of Proc2 */

PROC PROC3(H13, H4, I6, I1, I2, I3, I4, I5, FileA, FileB )
 BEGIN
 I1 := H13 + 4
/* I1 = 63  : Point 14 */
 If H13 < 60
/* true */
  I2 := 4
  I3 := I2 + 6
/* I3 = 10 : Point 15 */
 else
  I2 := 6
  I3 := I2 + 6
/* I3 = 12 : Point 16 */
 fi
 I4 := 3
 While I3 > 5
/* 5 iterations */
  I4 := I4 + 2
  I3 := I3 - 1
 od
 /* I4 = 13 : Point 17 */
 If I4 = 13 then
/* nested ifs */
```

```
 If H4 > 60 then
  I5 := 6
 else
  I5 := 8
 fi
/* I5 = 8 : Point 18  */
 I6 := 8 * I5
/* I6 = 64 : Point 19 */
 Put FileA, I6, 1
 I6 : = I6 * 2
/* I6 = 128 */
 Fetch I6, FileA, 1
/* I6 reverts back to 64 : Point 20 */
 if I6 = 64
  Put FileB, 'Success', 1
 else
  Put FileB, 'Failure', 1
 fi
end.

PROC PROC4(J1, J2, J3, J4, J5 )
 BEGIN
 J1 := 3
 J2 := 4
 J3 : = 6
 J4 := 4 * 2
/* J4 = 8 : Point 21  */
 J5 := J1 + J2 + J3 + J4
/* J5 = 21 : Point 22 */
 end.
```

```
PROC Proc5(I6, K1, FileA, FileC)
 BEGIN
 Fetch I6, FileA, 1
/* ties in with Proc3 */
 K1 := I6 + 4
/* K1 = 68 : Point 23 */
 Put FileC, K1, 1
 if K1 = 68 then
/* correct result */
  Put IOTerminal, 'Success', 1 /*  Point 24 – value of K1*/
 else
  Put IOTerminal, 'Failure', 1 /*  Point 24 – value of K1*/
  Put IOTerminal, 'Incorrect Value is ' + I6, 1
 fi
 While I6 < 100
  I6 := I6 + 10
 od
 Put FileA, I6, 1
end.
```

7.3.1 Procedural Concurrency

The normal values produced during a strictly sequential execution is the following:

Point	Name Of Variable	Sequentially-executed (correct) value	Actual value produced during Procedural Level Independence	Actual value produced during Individual Codeline Level Independence
1	A	7	7	7
2	B	30	30	30
3	D	54	54	54
4	E	45	45	45
5	F2	21	21	21
6	F	9840 (after iterations)	9840	9840
7	G2	27	27	27
8	G3	40	40	40
9	H4	38	38	38

10	H5	30	30	30
11	H8	20	20	20
12	H12	6	6	6
13	H13	59	59	59
14	I1	63	63	63
15	I3	10	10	10
16	I3	Error if reached	Not reached	Not reached
17	I4	13	13	13
18	I5	8	8	8
19	I6	64	64	64
20	I6	64	64	64
21	J4	8	8	8
22	J5	21	31	31
23	K1	68	68	68
24	K1	68	68	68

Table 7.3.1.1 Results of Independent Task Evaluation at the Procedural Granularity Level.

ProcName	Proc Seq No
Proc1	1
Proc2	1
Proc4	1
Proc5	1
Proc3	2
WSLCODELINESFORINDEPENDENTTASKS_VAR-INIT	1

Table 7.3.1.2 Independent Task Evaluation at the Procedural Granularity Level.

Thus, if the order of execution is changed as such:

```
Proc4(); // independent proc

Proc5(I6);

Proc2(H13, H4, I6);

Proc1(X1);
```

The results of the independent task execution, after the change in order of execution, of procedures confirm the correctness of the procedural level independent task evaluation algorithm.

7.3.2 Independent Tasks Evaluated at Individual Codeline Level

ProcName	Codeline	Codeline Seq No

	PROC WSLCODELINESFORINDEPENDENTTASKS_VAR-INIT(A, A2, A1, B, B1, B2, B4, C, C_TOP, C_INIT, D, E, X1, F, F1, F2, G1, G2, G3, G4, H1, H2, H3, H4, H5, H6,	
WSLCODELINESFORINDEPENDENTTASKS_VAR-INIT	H7, H8, H9, H10, H11, H12, H13, I1, I2, I3, I4, I5, I6, FILEA, FILEB, J1, J2, J3, J4, J5, K1, FILEC)	1
WSLCODELINESFORINDEPENDENTTASKS_VAR-INIT	CALL PROC1(X1, A, A2, A1, B, B1, B2, B4, C, C_TOP, C_INIT, D, E)	1
WSLCODELINESFORINDEPENDENTTASKS_VAR-INIT	CALL PROC2(H13, H4, F, F1, F2, G1, G2, G3, G4, H1, H2, H3, H5, H6, H7, H8, H9, H10, H11, H12, I6, I1, I2, I3, I4, I5, FILEA, FILEB)	1
WSLCODELINESFORINDEPENDENTTASKS_VAR-INIT	CALL PROC4(J1, J2, J3, J4, J5)	1
WSLCODELINESFORINDEPENDENTTASKS_VAR-INIT	CALL PROC5(I6, K1, FILEA, FILEC)	1
	PROC PROC1(X1, A, A2, A1, B, B1, B2, B4, C,	
PROC1	C_TOP, C_INIT, D, E)	1
PROC1	A := 1	1
PROC1	A2 := 4	1
PROC1	A := A1 + A2	2
PROC1	B := 4	2
PROC1	B1 := 6	2
PROC1	B2 := 3	2
PROC1	B4 := 8	2
PROC1	B := B1 + B2 * B4	3
PROC1	C := 4	3
PROC1	C_TOP := 8	3
PROC1	C_INIT := 4 + 5	3
PROC1	D := C_INIT * C	4
PROC1	C := C + 2	5
PROC1	OD	5
PROC1	IF (D > 50)	6
PROC1	E := D – C_INIT	7
PROC1	X1 := 36	7
PROC1	FI	7
	PROC PROC2(H13, H4, F, F1, F2, G1, G2, G3, G4,	
PROC2	H1, H2, H3, H5, H6, H7, H8, H9, H10, H11, H12)	1
PROC2	F := 3	1
PROC2	F1 := F + 4	2
PROC2	F2 := F1 * F	3
PROC2	F := (F + 1) * 3	4
PROC2	F1 := F1 -1	5
PROC2	OD	5
PROC2	IF F > 1000 THEN	6
PROC2	G1 := 9	7
PROC2	G1 := 12	8
PROC2	FI	8
PROC2	G2 := G1 * 3	9
PROC2	G3 := G2 + 3	10

PROC2	G4 := G3 + 6	11
PROC2	H1 := 3	11
PROC2	H2 := 4	11
PROC2	H3 := 6	11
PROC2	H4 := G3 + 8	12
PROC2	H5 := G2 + 3	13
PROC2	H6 := 9 * 1	13
PROC2	H7 := 8 + 1	13
PROC2	H8 := (8 * 2) + 4	13
PROC2	H9 := 12	13
PROC2	H10 := 3	13
PROC2	H11 := 4	13
PROC2	H12 := 3 + 3	13
PROC2	H13 := H9 + H6 + H4	14
PROC2	CALL PROC3(H13, H4, I6, I1, I2, I3, I4, I5, FILEA, FILEB)	15
PROC3	PROC PROC3(H13, H4, I6, I1, I2, I3, I4, I5, FILEA, FILEB)	1
PROC3	I1 := H13 + 4	1
PROC3	IF H13 < 60	2
PROC3	I2 := 4	3
PROC3	I3 := I2 + 6	4
PROC3	I2 := 6	5
PROC3	I3 := I2 + 6	6
PROC3	FI	6
PROC3	I4 := 3	6
PROC3	I4 := I4 + 2	7
PROC3	I3 := I3 – 1	8
PROC3	OD	8
PROC3	IF I4 = 13 THEN	9
PROC3	IF H4 > 60 THEN	10
PROC3	I5 := 6	11
PROC3	I5 := 8	12
PROC3	FI	12
PROC3	I6 := 8 * I5	13
PROC3	PUT FILEA, I6, 1	14
PROC3	FETCH I6, FILEA, 1	15
PROC3	IF I6 = 64	16
PROC3	PUT FILEB, %SUCCESS%, 1	17
PROC3	PUT FILEB, %FAILURE%, 1	18
PROC3	FI	18
PROC4	PROC PROC4(J1, J2, J3, J4, J5)	1
PROC4	J1 := 3	1
PROC4	J2 := 4	1
PROC4	J4 := 4 * 2	1
PROC4	J5 := J1 + J2 + J3 + J4	2
PROC5	PROC PROC5(I6, K1, FILEA, FILEC)	1
PROC5	FETCH I6, FILEA, 1	1
PROC5	K1 := I6 + 4	2
PROC5	PUT FILEC, K1, 1	3
PROC5	IF K1 = 68 THEN	4
PROC5	PUT IOTERMINAL, %SUCCESS%, 1	5
PROC5	PUT IOTERMINAL, %FAILURE%, 1	6
PROC5	PUT IOTERMINAL, %INCORRECT VALUE IS % + I6, 1	7

PROC5	FI	7
PROC5	I6 := I6 + 10	8
PROC5	OD	8
PROC5	PUT FILEA, I6, 1	9

Table 7.3.2 Results of Independent Task Evaluation at the Individual Codeline Granularity Level.

Independently-executing codelines, which have the same task sequence number, are re-arranged in random order to simulate a parallel execution. The results confirm the correctness of the specified individual codeline algorithm.

7.4 Evaluation of Proposed Independent Task Evaluation Method and Algorithms

One of the advantages of the specified approach is that the units of granularities selected, such as individual code line, procedure, and program block, are that some of the units may be used in modelling various UML diagrams. For example, individual code lines that may execute in parallel are modelled as parallel flows in the code-based activity diagrams. Procedures that may execute in parallel are given the same sequence number in sequence diagrams.

The disadvantages of the specified approach are several. Individual code line granularity, although needed to model flows in activity diagrams, entail too high a communication cost to outweigh the performance gain obtained through their parallelisation. Procedure granularity, although also useful in modelling UML diagrams, is not an optimal granular unit for parallelisation for two reasons. One reason is that due to the design of a typical batch-oriented system, procedures were designed to be called in a strictly sequential manner and efforts to overcome this design and to parallelise these procedures are difficult. The second reason is that the procedural granular unit is too large a unit to achieve optimal parallelism (Joiner, 1998).

Although years of software experience have validated the program block as the optimal granular unit in terms of size versus communication costs, identifying the optimal algorithm for determining program blocks is a difficult and lengthy process. Any such algorithm would be highly dependent on the underlying hardware platform. Communication costs, which are an important factor in determining the optimal granular unit in terms of granular unit size as determined by the algorithm versus communication costs, are highly dependent on hardware and its layout. An algorithm which has been determined to be optimal for one hardware platform would not necessarily be optimal for another hardware platform (Lam, 1992). Consequently, the identification of the proper granular unit size for parallel execution optimisation is outside the scope of this thesis.

7.5 Using Independent Task Evaluation Algorithms in Modelling Legacy Systems through UML Diagrams

Identifying independent tasks in the legacy system at the procedural and individual codeline granular level is an important step in adequately modelling a legacy system through UML diagrams. The generated UML activity diagrams model activities of the

legacy system at the individual code line level. Additionally, these sequence diagrams model procedure calls within and among objects. Therefore, in order to model parallel tasks in UML activity or sequence diagrams, it is first important to determine which tasks, whether at the individual codeline or procedure call granular level, may be executed independently or not. Independently executing task are modelled as parallel flows in UML activity or sequence diagrams.

Furthermore, identifying independent tasks is an important prerequisite for modelling events in UML diagrams. Events are modelled in the UML statechart and sequence diagrams. Events in UML diagrams are depicted as asynchronous or synchronous. In order to determine whether an event is synchronous or asynchronous, it is first necessary to determine whether the task immediately subsequent to the task where the event occurred may execute independently or whether this task must first wait for the event, with its subsequent event handler invocation, to finish executing. If the immediate subsequent task can execute independently, the event is depicted as asynchronous. However, if the immediate subsequent task must wait for the event, with its event handler, to finish executing first, the event is depicted as synchronous.

Identifying independent tasks at the program block level is also important for two reasons. One reason is that it provides a benchmark, an intermediate granular unit, to which to measure the degree of independent tasks discovered at both a fine, individual code line, and coarse, procedural, level of granularity. Another reason is that enables these identified program blocks, which were evaluated to be able to be executed independently, to be programmatically coded as independent tasks in a potentially future parallel-executing code generation tool (Chandi, 1992).

7.6 Event Identification

The final transformation step in transforming the original procedural-structured and sequential legacy system to an object-oriented, event-driven system is to identify possible events from source code. This event identification step is accomplished by constructing a control graph of procedure calls, I/O calls, system interrupts, and error invocations by parsing the source code. The nodes of this control graph that represent interaction with other objects are modeled as events. An example, an interrupt is modeled as an event occurrence between the object where the interrupt occurred and the System object, which handles interrupts. Once the events are identified, each event is evaluated in terms of its synchronicity. This evaluation is accomplished by evaluating, at the individual code line level of granularity, the task where the event occurred and the task immediately successive to this task in order to determine if any control or data dependencies exist among them. If a dependency exists, the event is deemed to be synchronous; otherwise, if no dependency exists, the event is deemed to be asynchronous.

Alhir defines an event as an occurrence that may trigger an object to change states. While events may be modelled as occurrences that trigger a transition between individual states, events typically are modelled as significant occurrences that affect an entity (Alhir, 1998).

In an event-driven system, events are handled regardless of when they occur. In a procedural-driven system, events are handled only if the procedure in which they occur is able to handle such an event. Because there is no straightforward way to transform a procedural-driven to an event-driven system, significant occurrences, such as a procedure call, in a procedural-driven system are modelled as pseudo-events during this transformation process, in order to distinguish them from events of an event-driven system.

The identification of pseudo-events, along with the determination of the asynchronous nature of these events, is important when modelling this system as UML activity diagrams. Events in activity diagrams must be identified and the nature of these events, whether they are asynchronous or synchronous must be specified. Furthermore, these events play a role when translating the system's UML representation to a potential C++ representation. In C++, events, along with their event-handling code, can be specified (Lippman, 1998).

Although any events identified from a purely sequential legacy system will be pseudo-events, this identification, along with their representation in UML or C++, allows the modelling of this system as an event-driven rather than a purely procedure-driven system as was the case previously. Furthermore, the constructs introduced to represent the pseudo-events can be used to represent future actual events in this system.

7.7 Development of Event Identification Method and Algorithm

After the legacy source code has been converted into a WSL representation, this WSL representation must be analysed, in turn, in order to identify pseudo-events.

One disadvantage of WSL is that WSL does not model interrupts well. The thesis' method of representing interrupts within WSL is to enclose a single WSL statement or small set of WSL statements within an if control construct and, in this if construct, test if a system-defined bit, System.Interrupt_Status in the case of system interrupts and System.Error_Status in the case of error interrupts, has been set or not. If this bit has been set, the appropriate system-defined handler procedure is invoked. In the case of the system interrupts, this handler is defined as the System_Interrupt procedure; in the case of an error interrupt, this handler is the System_Error procedure. Otherwise, if no interrupt occurs, the enclosed WSL statement(s) will be executed.

7.8 Identifying Pseudo-Events

Event-driven programming is ideal for programming user interfaces where the occurrence of tasks is undetermined and unpredictable. However, many legacy systems were programmed as batch processes with little user interaction and a clearly defined task to be executed in strict sequence. Reengineering a batch-oriented, procedural-driven legacy system, such as the telecommunication legacy system used in this case study, involves analysing this system for objects and their events. Some of the most common objects in batch-oriented systems are File, System, and Peripheral Devices. Events within this system could be interactions with these objects; these interactions could include file I/O operations and system calls.

The first step in identifying and representing these pseudo-events is to determine what constructs of a legacy system can be represented as pseudo-events. These constructs could be classified as the following:

- method invocations/procedure calls
- I/O functions (read/write functions to files)
- Interactions with external actors (keyboard, terminal)
- System calls, such as the Time() function
- Error handling
- System Interrupts

WSL is a procedural language while UML, in its statecharts, model events using objects. Consequently, any pseudo-event in WSL involves first looking up the objects associated with the procedures and variables in the relevant WSL statement and then constructing an event with these objects.

If any control constructs, such as if or while statements, enclose a pseudo-event, the conditions of these control constructs from guards for these events. An example, the WSL statements:

```
If (Acct-Balance < 0) Then
  Call Overdraft(Acct-Balance)
Fi
```

Is converted to its UML equivalent:

[ObjectA.Acct-Balance < 0**]/**Event_Overdraft(ObjectA.Acct-Balance) **SourceObject:** A, **TargetObject:** ObjectB.
The conditions of a while statement form a guard in the same manner as a WSL if statement except that a "*" character is appended at the end of the guard to indicate iteration.

An example of parsing WSL statements in order to identify pseudo-events is given as follows:

WSL representation:

```
If (System.Interrupt_Status <> 'Interrupt') And (System.Error_Status <> 'Error') Then
 /* checks to see if system interrupt or error interrupt has occurred */
While  (AcctsFile.File_Status <> EOF) do  /* while control construct */
   If (System.Interrupt_Status <> 'Interrupt') And (System.Error_Status <> 'Error') Then
                                Fetch AcctRec, AcctFile, AcctFile_Index    /* I/O (read) function */
      Elif  (System.Error.Status = 'Error') Then
    System_Error()
      Else
    System_Interrupt()
      fi
Od
```

```
If (System.Interrupt_Status <> 'Interrupt') And (System.Error_Status <> 'Error') Then
  If (Acct-Balance > 0) Then  /* if control construct */
   Call Acct-Overdraft(Acct-Number, Acct-Balance)        /* procedure call */
  Fi
    Elif (System.Error_Status = 'Error') Then
      System_Error()
    Else
  System_Interrupt()
     fi
If (System.Interrupt_Status <> 'Interrupt') And (System.Error_Status <> 'Error') Then
  Fetch CurTime, System_Time  /* system call */
    Elif (System.Error_Status = 'Error') Then
   System_Error()
    Else
  System_Interrupt()
     fi
If (System.Interrupt_Status <> 'Interrupt') And (System.Error_Status <> 'Error') Then
 Fetch CurInput, IOTerminal  /* interaction with external actor such as a keyboard */
    Elif  (System.Error_Status <> 'Error')  Then
       System_Error()
    Else
  System_Interrupt()
     fi
If (System.Interrupt_Status <> 'Interrupt') And (System.Error_Status <> 'Error') Then
    If NOT (Acct-Number <= 0) Then
      System_Error("Invalid Account Number")
    Fi
     Elif (System.Error_Status = 'Error') Then
      System_Error()
    Else
 System_Interrupt()
```

fi

Elif (**System.Error_Status** = 'Error') **Then**

System_Error()

Else

System_Interrupt()

Fi

Their corresponding representation as events in UML:

[ObjectA.AcctFile.File_Status <> EOF]*/Event_ReadAcctRec(ObjectB.AcctRec, ObjectA.AcctFile, ObjectA.AcctFile.File_Index) SourceObject: ObjectA, TargetObject: ObjectB

[ObjectA.Acct-Balance<0]/Event_CallAcct-Overdraft(ObjectA.AcctNumber, ObjectA.Acct-Balance) SourceObject: ObjectA, TargetObject: ObjectB

Event_ReadCurTime(ObjectA.CurTime) SourceObject: ObjectA, TargetObject: System

Event_ReadCurInput(ObjectA.CurInput) SourceObject: ObjectA, TargetObject: IOTerminal

[ObjectA.Acct-Number <= 0]/Event_SystemError("Invalid Account Number") SourceObject: ObjectA, TargetObject: System

[System.Interrupt_Status = 'Interrupt']/Event_SystemInterrupt() SourceObject: ObjectA, TargetObject: System

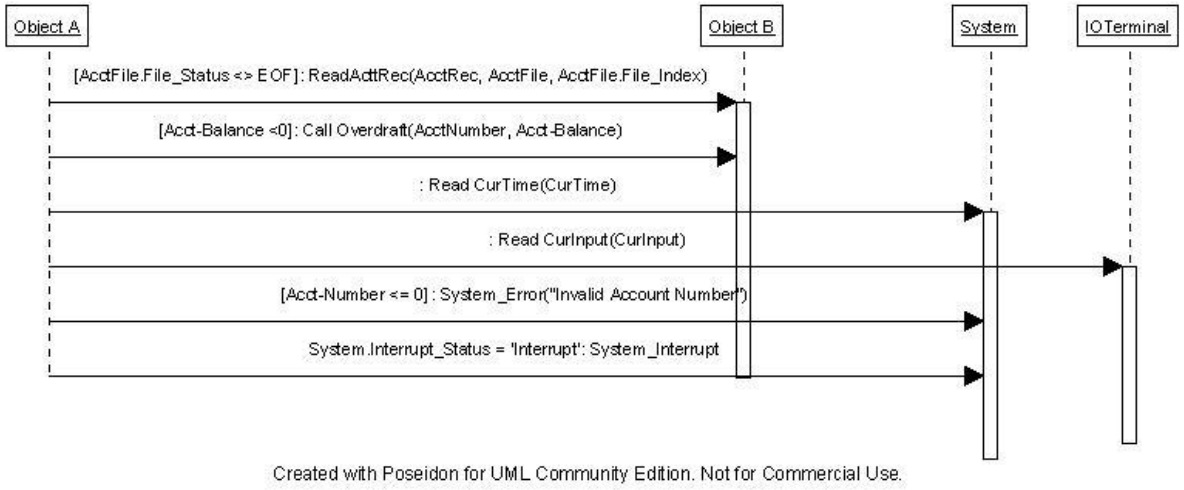


Fig 7.8.1.1 Sequence Diagram of Pseudo Events from Sequence Code.

There are a number of system-defined external objects in the generated UML representation of this system. One external object is the *System* object that handles both system and error interrupts and has, as its methods, *System_Interrupt* and *System_Error*. System and error interrupts are modelled in UML as events between the class in which the interrupt occurred and the *System* object. Another external system-defined object is the *File* object which has as its methods, *Fetch* and *Put*. Calls to *Fetch* and *Put* are modelled in UML as events between the class in which they are invoked and the *File* object.

In this particular COBOL legacy system, the present error handling is minimal to non-existent. Additionally, this COBOL legacy system has no explicit system interrupt handling. When this COBOL legacy system was converted into a WSL representation, each WSL statement, or set of WSL statements, is enclosed by an if control construct that, when executed, tested the system-defined status bits, System.Interrupt_Status and System.Error_Status, to determine if an interrupt has occurred. If an interrupt has not occurred, the enclosed WSL statement executes normally; otherwise, the appropriate interrupt handler is invoked.

One of the problems with enclosing each converted WSL statement with an interrupt test and handler is that it increases the size of the WSL program by a factor of five (the original WSL statement + interrupt test statement + error interrupt test + error handler invocation statement + system interrupt handler invocation statement). Modelling the interrupt test statements, as guards for events, and interrupt handler invocation statements, as events, in UML would unnecessarily complicate and clutter the UML activity and statechart diagrams generated from this WSL representation. Consequently, the interrupt test and interrupt handlers in WSL are ignored as guards/events when converting a WSL representation to a UML activity diagram. Instead, system and error interrupts are modelled as single events between each internal class in the system. Internal classes are classes that were defined through the specified object clustering method of variables and procedures of the legacy system (Millham, 2002). An example, there are twenty classes in this system. System interrupts are modelled as single events between each internal class in the system and the external *System* object. Error interrupts are modelled in a similar way, one error interrupt per internal object of the system.

Following these rules of pseudo-event identification, which involve procedural calls, I/O functions, et al, the legacy source code is analysed to extract these possible pseudo-events along with their context. This extraction of pseudo-events occurs after the task parallelisation/sequentialisation process has completed. Consequently, each pseudo-event can be analysed in terms of its independence or dependence on tasks.

In order to determine whether or not the event is synchronous or asynchronous, it is necessary to first determine if the immediately succeeding tasks are dependent on the pseudo-event completing its execution before proceeding. If such a dependency exists, then this event is asynchronous and the number of tasks which are dependent on this pseudo-event completing its execution determines this asynchronous event's degree of asynchronicity. In other words, the degree of asynchronicity is the number of sequential task groupings that can occur before a dependency with the pseudo-event occurs. For example, suppose a pseudo-event has a task sequence number of 4 and the next possible control or data dependency occurs in task sequence number 7, this pseudo-event will have a degree of asynchronicity of 3 task sequence groupings. Each grouping consists of one or more tasks. If multiple tasks are in the grouping, each task must be able to execute independently.

This experiment is an investigation into the nature of events or pseudo-events within a legacy system. Notably, an attempt is made to determine if converting a previously procedure-driven, sequential system to an object-oriented, event-driven, distributed computing system results in a true transformation of the legacy system or whether the remodelled system retains the structure of the previous legacy system, namely that a majority of its procedure/system calls and I/O operations remain synchronous.

Because independently-executing tasks are grouped into task sequence groups with sequentialisation of tasks due to some data/control dependency between task sequence groupings, this grouping transforms the previously sequential system to a system capable of running properly in a distributed computing environment.

Furthermore, this grouping ensures that a mere reordering of codelines can not alter the degree of asynchronicity of events.

7.9 Sequence Diagrams

Sequence diagrams are diagrams that depict the interactions, via message passing, among objects in a temporal context. The sequence diagram is divided up into vertical sections, called lifelines. Each object in the system is assigned their own swimlane or partition. Messages are depicted as being sent from the swimlane, or partition, of their sending, or source, object to their receiving, or target, object.

Procedure calls are modelled as messages between the caller object that invokes the procedure and the called object that contains the procedure being invoked. Exceptions and interrupts are modelled as messages between the object where the interrupt/exception occurred and the System object, representing the underlying operating system, which handles the error or exception. File input/output calls, such as statements that read or write data from files, are modelled as messages between the object invoking the input-output method and the File object, which represents a generic database file.

Conditions within WSL control constructs, such as WSL's if-then statements, that enclose code that invoke a message form conditions within the guards that govern the passing of messages from one object to another. An example, given the WSL statement, **IF W<3 THEN Call UpdateVal FI**, the condition **[W<3]** forms a guard to the method invocation message **UpdateVal**. If the WSL control construct that encloses code invoking a message is a do-while loop, the guard condition parathenses have a ***** suffix, in the format [<Condition>]*.

These messages may be modelled as synchronous or asynchronous. A message is deemed to be asynchronous if the task that is immediately successive to the task invoking the message may execute independently. If the immediately successive task can not execute independently, the message is deemed to be asynchronous. By evaluating tasks containing message invocations, the independent task evaluation process determines which tasks may execute independently from one another and, consequently, determines which messages contained in the tasks being evaluated are deemed to be synchronous or asynchronous.

The independent task evaluation process is responsible for determining the sequence numbers assigned to each message in the sequence diagram. The independent task evaluation process assigns a sequence number to tasks at both the procedural and individual codeline level of granularity.

Each message depicted in the sequence diagram is given a sequence number in the number: <procedure task sequence number>.<individual codeline task sequence number>. The procedure task sequence number is the sequence number of the procedure where the message is invoked and individual codeline task sequence is the sequence of the codeline where the message

is invoked. The sequence indicates the order of execution in ascending order; messages with the same sequence number may be executed in parallel.

7.10 Extraction Method of Sequence Diagrams

The sequence diagram extraction algorithm is based on information gained during the event identification, object identification, and independent task evaluation reengineering processes. Messages are modelled from identified events. Inter-class (between different classes) events are modelled only in the sequence diagram as a dynamic relationship between two classes. Intra-class (within the same class) are not modelled in this diagrams as these events do not model relationships between two different classes. Events are often, in batch-oriented systems, system interrupts or errors, file operations, or procedure calls. Each class involved in the sequence diagram is allocated its own swimlane or partition in the diagram.

The following algorithm is used:

i. For each identified class object, model two messages between this class (source object) and the generic System (target object) object class that represents the underlying system platform. These messages model the System Interrupt and System Error events for these identified classes. All of these messages are modelled as asynchronous messages with no particular sequence number.

ii. For file operation events, first determine the class encompassing the procedure that contains the file operation code line. The file operation is modelled as a message between this class object (the source class) and the generic File (target class) class object. The destination, index, and source variables of the file operation form the message parameters. If the codeline of the file operation can execute concurrently with its immediately successive codelines, as determined by the individual codeline independent task evaluation algorithm, this message is modelled as an asynchronous message; otherwise, it is modelled as a synchronous message. The execution sequence numbers of the file operation codeline and its procedure form the message sequence number in the format <Procedure_Sequence_Number>.<Codeline_Sequence_Number>.

iii. For procedural calls, first determine the classes encompassing both the caller procedure (source object) and the called (target object) procedure. The procedural call is modelled as a message between these two class objects with the parameters of the procedural call forming the parameters of the message. If the codeline containing the procedural call can execute concurrently with its immediately successive codelines, as determined by the individual codeline independent task evaluation algorithm, this message is modelled as an asynchronous message; otherwise, it is modelled as a synchronous message. The execution sequence numbers of the procedural call codeline and its enclosing procedure form the message sequence number in the format <Procedure_Sequence_Number>.<Codeline_Sequence_Number>.

7.11 Sample of WSL Code to Sequence Diagram

A small sample of WSL code is provided alon with the corresponding UML sequence diagram based on this code.

WSL Code Sample:

```
Class A
 Begin
  VAR NewElement1
  VAR OldElement1
  VAR IndexElement1
  PROC P1( )
   Begin
    CALL P2()
    PUT NewElement1, OldElement1, IndexElement1
   End

Class B
 Begin
  PROC P2( )
   Begin
   End
 End
End
```

The following diagram models the preceding WSL code sample as messages between classes in a sequence diagram. Each inter-class procedure call is modelled as a message between the caller class and the calling class. Each I/O file operation is modelled as a message between the class where the operation is invoked and the File class object. Any parameters of these operations or calls are included in the parameter list of the message in the sequence diagram.

Fig 7.11.1 Sequence Diagram Modelled from WSL Code Sample.

7.12 Summary

Sequence diagrams are needed by developers in order to understand how objects interact amongst each other. This importance is highlighted in two ways. One way is that these objects are not developer-designed objects; hence, the developer has no way of knowing, other than from a sequence diagram, how one reengineered class calls another. The other way is that when a legacy system is moved into a new platform, such as an object-oriented, Web-based, platform where objects may be scattered over several machines, knowledge of their interaction, as depicted in the sequence diagram, is key in order to prevent unwanted side-effects as classes interact from one machine to another.

Evaluating and determining independent versus sequential tasks, at the source code level, using tasks at both and at the procedural and codeline level of granularity, is important in the extraction of UML diagrams. In order to restructure these legacy systems correctly, restructuring algorithms, such as the independent task evaluation aglorithm, must be validated, in this case through the use of a series of carefully selected test cases. Events, that are identified within the source code and are encapsulated within tasks, must have their task execution independence determined in order to model these events, in their proper sequence, within the UML sequence diagrams.

Independent task evaluation also later plays a role in the extraction of UML activity diagrams where flows to activities/action states are modelled as parallel or sequential depending on the results of the evaluation of the enclosing and immediate successive task of the activity/action states as to their task independence. Parallel executing tasks are modelled as parallel flows; sequential executing tasks are modelled as sequential flows. Object identification, event identification, and independent task evaluation processes are also used in UML sequence diagrams where the identified events are modelled as messages between classes whose message sequence is determined by the independent task evaluation algorithms. Consequently, there is a direct correlation between the information gained during the restructuring of the system and its use in the extraction and modelling of UML diagrams from legacy system source code.

Chapter 8: Independent Task Identification and Activity Diagram Extraction

8.1 Introduction

This chapter introduces a method to extract activity diagrams from source code. In order to extract these diagrams, activities represented in the activity diagram must be identified. Because these activities are based, for the most part, on individual codelines, the reengineering tool must know which tasks, identified as individual codelines, must be able to execute independently and which tasks must execute sequentially. In this way, parallel executing activities are modelled as parallel flows in the activity diagrams; sequentially-executing activities are modelled as sequential flows. Consequently, an algorithm to evaluate the independence of execution of individual codelines is provided in this chapter. Additionally, several algorithms to extract these activity diagrams are needed in order to handle branching within, linking activity sequences, and simplifying activities through pattern-matching.

8.2 Activity Diagrams

Activity diagrams describe the internal behaviour of a class method as a sequence of steps. These sequence of steps model the dynamic, or behavioural, view of a system in contrast to class diagrams, which model the static, or structural, view of the system.

An activity in UML represents a step in the execution of a business process. Activities are linked by connections, called transitions, which connect an activity to its next activity. The transitions between activities may be guarded by mutually exclusive Boolean conditions. These conditions determine which control flow(s) and activities are selected (Muller, 2000a).

Activity diagrams may contain action states. Action states are states that model atomic actions or operations. Activity diagrams may also contain events (Alhir, 1998).

Activity diagrams can be partitioned into object swimlanes, also called partitions, that determine where an activity is placed in the swimlane of the object where the activity occurs (Alhir, 1998). Activities may be attached to a class, an operation, or a use case (Oestereich, 1999).

One might question why TAGDUR chooses to generate activity diagrams from WSL code rather than simply allow the developers to view the WSL or potentially generated C++ code of the transformed system. Activity diagrams were chosen for many reasons. UML is widely understood by many developers while WSL and, to a much lesser extent, C++ has less of a universal understanding. Furthermore, activity diagrams clearly represent the interaction among objects and the occurrences of events among activities; this representation would be much less apparent than if the developer were simply perusing the code of the transformed system. Although the specified activity diagram is based on WSL code and individual WSL code lines are used to distinguish action states in the activity diagram, this lack of understanding is mitigated by attached comments which describe the

WSL code line being modelled. An example, a file I/O event is described in terms of its type (File I/O), sub-type (Read), and destination, source, and index variables. The decision to base this generated activity diagram on WSL, rather than C++, was due to several reasons, including the fact that WSL is programming and platform independent. An example, a file I/O operation is represented through one type of WSL statement while representing the same operation in C++ may take several C++ statements depending on the type of file being accessed. Consequently, many implementation-specific details that would clutter an activity diagram based on C++ code are avoided if this same activity diagram is based on WSL code instead.

Another advantage in the parsing of the WSL source code in order to obtain information that can be used to derive activity diagrams, such as the type of operation and operands involved in each code line, is that this information will be available in TAGDUR's database for use during the generation of business rules, as a possible future extension to the TAGDUR tool.

8.3 Development of Independent Task Evaluation Algorithms

In order to accurately depict parallel or sequential transitions between activities/action states in activity diagrams, independent tasks, or their counter-part sequential tasks, must be identified. Tasks that have been determined to be able to execute in parallel by the independent task evaluation step of the transformation process are modelled as parallel activities and flows in the activity diagram while tasks that have been determined to be able to execute sequentially only are modelled as sequential activities and flows. In activity diagrams, synchronisation bars are used to synchronise the divergence of sequential activities into parallel tasks or the merging of parallel tasks to a sequential task. These enable the control flow to transition to several parallel activities simultaneously and to ensure that all parallel tasks complete before proceeding to execute the next sequential task (Muller, 2000a). Conditions within WSL control constructs, such as WSL's if-then statements, form conditions within the guards that govern the flow of control to activities enclosed by the condition blocks of this WSL control construct.

In these independent algorithms, we evaluate tasks at the individual codeline, procedure, and program block level of granularity. Because the WSL code lines of a procedure form steps in the execution of this procedure and because individual WSL code lines form an atomic unit of execution, basing activities on individual WSL code lines is a logical basis for the nodes of an activity diagram. Similarly, basing tasks at the procedural level of granularity enables us to determine whether procedure calls, modelled in a UML sequence diagram, are modelled as synchronous messages or not.

In order to identify independent tasks, Eisenbarth's method, which employed static call analysis to identify data and control dependencies between procedures, was used for procedural tasks (Eisenbarth, 2001). Gall identifies procedures as entities that incorporate the highest level of atomic control in a program (Gall, 1995). Similarly, static analysis of the code to determine data and control dependencies between successive code lines or program blocks was used to indentify independent tasks.

The extraction method for activity diagrams is based on the parsing of individual WSL codelines and their analysis. Through parsing, each WSL codeline is associated with the procedure that it belongs to. The control blocks that a WSL codeline may be associated with are identified; the identifying information regarding this control block (such as if/while level numbers [to indicate nesting] and if/while sequence numbers [to distinguish control blocks with the same level of nesting]) are recorded for use by the

independent task identification algorithm. It is necessary to identify these control blocks in order to identify control dependencies. Control, along with data, dependencies are used to determine if two successive WSL codelines could be executed in parallel or if they must be executed sequentially. Furthermore, two successive control blocks, identified through this method, could also be used to determine if these program blocks could execute independently or if they must execute sequentially. If two successive WSL codelines have a control dependency, the transition from one WSL code line (modelled as an action state/activity) to another is modelled as a guarded transition in the resulting activity diagram. Similarly, parallel executing WSL codelines are modelled as parallel flows in this activity diagram.

8.4 Algorithms

This thesis' algorithms, whether or not these algorithms evaluate potentially parallel tasks or handle conditional branching, are based on a partitioning of the activity diagrams into procedure activity diagrams, where each activity diagram represents the behavior of activities within a procedure, and a package activity diagram, which represents the activity diagram of the main controller procedure of the package calling the various procedures. In this way, procedure calls can be expanded to the full procedure activity diagram or collapsed into a single sub-machine activity. The package activity diagram is attached to the Package element. In this way, the procedural flow of control of the package, or software component, is modeled (OMG, 2004: p 3-156).

By partitioning activity diagrams into procedures, the problem of an explosion of action states as one tries to model one procedure calling another recursively is avoided. Furthermore, this partitioning greatly simplifies the diagram and aids the developer's understanding of the system behavior. If the developer wishes to view the activities within a procedure, they may select that procedure's activity diagram. If not, this procedure call is represented by a single submachine state in the caller procedure's activity diagram.

An action state is a state with a single entry state and an implicit exit transition upon completion of an activity. Each action state of the activity diagram represents usually a single autonomous activity. The action represented by the action state may be represented in pseudo code, natural language, or programming language. The execution of action states may be governed by guard conditions (OMG, 2004, 3-159). In the generated activity diagram, the action state is represented by an individual codeline, which represents an autonomous action, of WSL.

In order to derive the activity diagram from the source code, a number of steps must be taken:

1) Step1: represent the WSL code in a table with fields to represent the codeline, the codelineID (unique ID of that codeline), execution sequence numbers, while and if sequence numbers and levels, and the procedure name that the codeline belongs to

2) Step 2: determine the while and if sequence numbers and levels of each codeline in a procedure (see Algorithm 8.3.1 Step 2)

3) Step 3: evaluate the independent task execution of individual activities within a procedure. Sequence numbers, from 1.. n, indicate the order of execution of individual code lines. Independent tasks (activities of independently executing tasks) are denoted with the same sequence numbers meaning that they can be executed concurrently; sequential tasks, which must execute sequentially, are given different sequence numbers in ascending order of their execution.

4) Step 4: simplify and describe the operations of the activities

5) Step 5:determine the branching, and consequent guards, of each activity diagram

6) Step 6: create the transitions for the activity diagram.

Step 1 was conducted, with empty values in the tables for if/while sequence numbers, but with the procedure name and individual codelines during earlier parsing of the WSL source code.

8.4.1 Step 1: Algorithm to Represent WSL Code

The purpose of this step is to incorporate each WSL statement, along with the name of its procedure, into the database so that further analysis of this code could proceed.

For each WSL code line (in sequential order)

a. Parse each WSL code line to identify when the procedural section of the WSL representation begins and to identify the procedure that the WSL codeline being parsed belongs to

    i. The procedural section of the WSL representation begins after the variable declaration section. The variable declaration section has each of its code lines begin with a **VAR** or **VAR STRUCT** keyword. Following the variable declaration section, the procedural declaration section begins. The procedural declaration section begins with the listing of the WSL procedures. Each procedure declaration line begins with the **PROC** keyword.

    ii. Procedural declarations are in the format **PROC** <ProcedureName>**(**<parameter list>**)**. If a WSL codeline that is being parsed begins with the **PROC** keyword, then it is a procedural declaration. Hence, extract the ProcedureName and this procedure name becomes the current procedure of subsequent WSL codelines being parsed until the **.END** keyword is encountered during parsing which denotes the end of that procedure

b. If the WSL code line is not a variable or procedural declaration, insert the line and the procedure name that this line belongs to into the database into its own special table with empty execution sequence numbers and while and if sequence numbers and levels for use by later algorithms.

8.4.2 Step 2: Algorithm to Determine While/If Sequence and Level Numbers

The purpose of the If/While Level numbers is to distinguish the individual codeline's level of nesting of if/while blocks within a procedure. The If/While Sequence numbers distinguish codelines that have the same level of nesting but belong to different if/while blocks within the same procedure.

For each procedure

a. Initialise if/while levels and sequence numbers to zero

   i. IfLevelNo – represents the if level of current codeline within the current procedure. The outermost if block is represented with the IfLevelNo = 1, the second outermost if block is represented with the IfLevelNo = 2, and so on

   ii. WhileLevelNo – represents the while level of the current codeline within the current procedure. The outermost while block is represent with the WhileLevelNo = 1, the second outermost while block is represented with the WhileLevelNo = 2, and so on

   iii. IfSeqNo – represents the sequence number of if blocks within the procedure. The first if block has a IfSeqNo = 1; any nested if blocks within this first block have the same IfSeqNo of 1. The second if block that is not nested within this first block has the IfSeqNo of 2. Any nested if blocks of the second if block had the IfSeqNo of 2.

   iv. WhileSeqNo – represents the sequence number of while blocks within the procedure in a similar manner to IfSeqNo

b. From the first code line to the last line of the current procedure, go through each codeline (referred to as current codeline)

   i. If an IF <Condition> is encountered in the current codeline, first increment IfSeqNo by one if IfLevelNo is zero (indicating an outermost if block) and then increment IfLevelNo by one.

   ii. If an WHILE <Condition> is encountered in the current codeline, first increment WhileSeqNo by one if WhileLevelNo is zero (indicating an outermost while block) and then increment WhileLevelNo by one.

   iii. If a **fi** keyword is encountered which denotes the end of an if block, decrement IfLevelNo by one

   iv. If an **od** keyword is encountered which denotes the end of a while block, decrement WhileLevelNo by one

   v. Update current codeline's IFLevelNo, IfSeqNo, WhileLevelNo, and WhileSeqNo in the table.

8.4.3 Step 3: Algorithm to Determine Concurrency or Sequential Activities

Because the generated activity diagrams are code-based and activities, for the most part, are based on individual code lines, these individual code lines must be evaluated in order to determine which individual code lines may execute in parallel; the parallel-executing codelines are modelled as parallel flows in the code-based activity diagrams.

This algorithm, like all levels of granularity, use a common function Check_For_Shared_Vars which has two parameters, one parameter is one set of codeline(s) and the other parameter is another set of codeline(s). If one set of codeline(s) updates one or more variables that are currently being used with the other set of codeline(s), then the function returns true; otherwise, the function returns false.

The algorithm to check individual programming codelines is as follows:

For every procedure

a. current task sequence number is initialised to 0;

b. For every code line (curcodeline) of current procedure

I) If curcodeline or next code line is a While or if construct, then update curcodeline's task sequence number to current task sequence number; increment to the next current task sequence number; update next code line's task sequence number to current task sequence number;

II) Otherwise,

If Check_For_Shared_Variables(curcodeline, next code line) returns true, then these two tasks are not independent tasks so update next code line to next task sequence number;

Otherwise, {no shared variables so these two tasks are independent};

Update curcodeline's task sequence number to current task sequence number;

Update next code line's task sequence number to current task sequence number.

Each code line is evaluated in context of its procedure rather than in context of its program. The reason for this evaluation is that procedures are a logical entity to encapsulate code lines and using this method, each code line is evaluated once, within the context of its procedure. If the code line was evaluated on a program-level basis and in relation to the procedural call graph, a code line may be evaluated several times.

If a control construct is encountered in a code line, the task sequence number is incremented to the next task sequence number. The reason for this incrementation is that any subsequent code lines that are enclosed by this control construct can not be evaluated in parallel with the control construct code line. The control construct code line must be evaluated first and then any enclosed code lines can be evaluated.

The maximum number of program blocks of the same level is selected in order to maximise the number of granular units that can be considered for parallelisation within a procedure.

8.4.4 Step 4: Algorithm to Simplify Code Use

This algorithm simplifies the code for later use by algorithms to determine branching and to produce action states from individual codelines. Additionally, for developers unacquainted with WSL, it produces a simplified description of the operation as well as any operands. Unless an operation is deleted or simplified, the whole WSL codeline forms the action of the action state of the codeline to which it represents. The description of the operation is just an additional aid to understanding for the developers.

1. For each procedure

1.1 For each codeline (current codeline of the procedure)

1.1.1 if current codeline is a comment or consists of the keywords: **od** (end of while condition block), **fi** (end of if condition block), delete codeline. This ensures that non-meaningful, codelines that serve little purpose other than comments or demarcation of condition blocks are eliminated. Also, procedure declarations are deleted.

1.1.2 if a current codeline is a branch (either if or while statement), ignore it. It will be used later by the other algorithms that determine and handle branching

1.1.3 if a current codeline is a simple assignment statement, which has the format <Destination_Variable> := <Operand><Operation><Operand>, extract the destination variable and describe the operation type as “Assignment”.

1.1.4 if a current codeline is a procedural call, in the format **Call** <ProcedureName>; extract the procedure name and describe the operation type as “Method Invokation”. The procedure name is described as CalledMethod.

1.1.5 if a current codeline is a system call such as a call to the System date function to obtain the system date, the operation type is described as “System Call”

1.1.6 (Step 4): if current codeline is a file I/O operation statement, in the format [Put | Fetch] <Destination_TopLevelStructureVariableName.ElementName>, <Source_TopLevelStructureVariableName.ElementName>, <Index_TopLevelStructureVariableName.ElementName>, is encountered, the top level structure names of the destination, source, and index variables are extracted. The operation, if a Put, is described as a Write or, if a Fetch, it is described as a Read. The top level structure names of the destination, source, and index variables are inserted into the table as Destination Variable, Source Variable, or Index Variable respectively. The succeeding codelines of the current procedure are examined. If this file I/O statement is followed by immediately successive file I/O statements which have the same top level structure names of the destination, source, and index variables but different Element Names, these statements are removed because they indicate that they are part of the same high level record operation. When a file I/O statement with different top level structure variables, a different operations statement, or the end of the procedure is encountered, this deletion process ends because it indicates that this high level record operation has ended. The purpose of this deletion process is to remove unnecessary detail from the activity diagram – the developer can see that record A was written to record B without viewing all of the elements that these records contain.

8.4.5 Step 5: Algorithm to Determine Branching in Activity Diagrams

In order to handle branching within an activity diagram, either as a result of an if or while statement, an algorithm is needed to model the branching, as encompassed as a guard attached the transition going to the first successive codeline within the condition block, in an activity diagram. In this algorithm, a simple table is used to record the branch condition, the first codeline activity of the succeeding if/while block (if the guard is evaluated to be true), and the first successive codeline or activity outside this if/while block (if the guard is evaluated to be false).

The first successive codeline or activity of this if/while block (if the guard is evaluated to false) is called the FirstBlockCodeline while the first successive codeline or activity outside this if/while block is called BranchOutCodeline. The method that is used to determine the BranchOutCodeline is similar to the backpatching method used in handling branching in many programming language compilers (Aho, 1986).

1. For each procedure

1.1 For each codeline (current codeline) of the current procedure in ascending order of original sequence

1.1.1 If current codeline is a non-branch (not an if or while) statement, ignore this statement

1.1.2 If current codeline is a branch statement, in the format [**If** | **While**] <Condition> [**Then** | **Do**], extract the condition and then use this condition to form the guard in the format **[**<Condition>**],**in the case of an if statement, or in the format **[**<Condition>**]*** in the case of a while statement. The ***** suffix to the guard denotes an iteration condition. Obtain the IfLevelNo, IfSeqNo, WhileLevelNo, WhileSeqNo settings of the current codeline branch – these variables become the BranchIfLevelNo, BranchIfSeqNo, BranchWhileLevelNo, and BranchWhileSeqNo variables respectively. Set flag of BranchFind to true. In the case of an if statement, set IfFlag to true; otherwise set it to false to indicate a While statement.

1.1.2.1 If FirstBlockCodeline is empty (indicating no succeeding branch to statement has been found yet), the BranchFind flag is true, and current codeline has same IfLevelNo and IfLevelSeq, in the case of an if statement, or WhileLevelNo and WhileLevelSeq, in the case of a while statement, of the newly-encountered branch statement of 1.1.2, set FirstBlockCodeline to the codelineID of the current codeline. Set BranchFind flag to false.

1.1.2.1 If the BranchOutCodeline is empty (indicating no succeeding branch to statement has been found yet) and the current codeline's IfLevelNo = (BranchIfLevel -1), if current codeline's IfLevelNo > 0, or IfLevelNo = 0 if BranchIfLevelNo is 1, and IfSeqNo = (BranchIfSeqNo), then first codeline outside the branching condition block has been found so set BranchOutCodeline to the codelineID of the current codeline. Similarly, with WhileLevelNo and WhileSeqNo in the case of a while branch.

1.1.2.2 If BranchOutCodeline is empty, meaning that the last codelines of the procedure are encompassed within a condition block, set BranchOutCodeline to -1 to indicate that the last activity will be the last activity of the procedure. Insert the codelineID (unique id of each codeline) of the branch

statement, the extracted condition in the proper format, the codelineID of the FirstBlockCodeline, and the codelineID of the BranchOutCodeLine to the BranchCondition table

8.4.6 Algorithm To Produce Activities or Action States from Code

This algorithm produces action states to represent atomic WSL operations. The action states are partitioned, as inclusion in an activity diagram, by packages (which represent individual COBOL source code files) and by procedures. One reason for this partitioning by procedures and packages is to retain the original programmer logic of cohesion in which they included certain operations within procedures. Other reasons include the simplicity of this algorithm where a method invocation can be represented by a CallState or, in further detail, by an activity diagram, representing the operations enclosed within the procedure.

The first operation is to put the codelines into a separate table with their CodelineID, Codeline, If/While Level and SeqNo, BranchOutCodeLine, Condition, FirstBlockInCodeline, and Sequence Numbers. Non-operational WSL codelines such as if-then, while-od, or comments are removed from this table for easier operation.

The next operation is to describe the WSL operation. In the case of a WSL I/O operation, the top level structure name of the index, destination, and source variables are extracted. Any immediately successive codelines with the same top level structure names for its index, destination, and source variables are removed until one of the following conditions is met

i) the end of the procedure is reached

ii) a file I/O operation with different top level structure names for its variables is reached

iii) a non file I/O operation is reached

8.4.7 Step 6: Algorithm to Produce Transitions

This algorithm partitions activity diagrams into a separate activity diagram for each procedure. By partitioning activity diagrams by procedure, method invocation can be represented as a call to a sub-activity graph, which represents the activity diagram of the invoked procedure. This method avoids the problem of indeterminate space explosion of action states where a procedure may repeatedly call a procedure and this repeated invocation is represented as an ever-increasing expansion of action states.

1.For each procedure

1.1 Create both a Start and End state for this procedure

1.2 CurState <- first codeline of procedure

1.3 CurStateSeqNo <- sequence of of first codeline of the procedure

1.4 CurStatePrevious <- Start state

1.4.1 Update codeline's field, Transitioned, to 0 in the procedure // initialize all codelines

1.5 For each codeline, in sequential order of execution, of procedure whose TaskSeqNumber = CurStateSeqNo and whose field, Transitioned, is 0 // not-handled

1.5.1 If codeline is a non-branch codeline

1.5.1.1 ParCntIncoming <- number of Incoming codelines in procedure with same sequence number of CurState (CurStateSeqNo)

1.5.1.2 If ParCntIncoming = 1 then // no concurrent incoming states

1.5.1.2.1 IncomingStatesList <- CurState

1.5.1.2.2 Else If ParCntIncoming > 1 then // concurrent incoming states

1.5.1.2.2.1 IncomingStateList <- list of codelines in procedure with same sequence number as CurStateSeqNo

1.5.1.2.2 ParCntOutgoing <- number of Outgoing codelines in procedure with same sequence number of CurState plus 1 (CurStateSeqNo + 1)

1.5.1.2.3 If ParCntOutgoing = 0 // end of procedure reached, so outgoing state is End State

1.5.1.2.3.1 OutgoingStateList <- End State of procedure

1.5.1.2.4 else if ParCntOutgoing = 1 // single outgoing state

1.5.1.2.4.1 OutgoingStateList <- outgoing codeline in procedure with same sequence number of CurStateSeqNo + 1

1.5.1.3 if (ParCntIncoming =1) AND (ParCntOutgoing <=1) then // determine Transition Type

1.5.1.3.1 TransitionType = ‘Simple’

1.5.1.4 else if (ParCntIncoming > 1) AND (ParCntOutgoing = 1) then

1.5.1.4.1 TransitionType = ‘Merge’

1.5.1.5 else if (ParCntIncoming = 1) AND (ParCntOutgoing > 1) then

1.5.1.5.1 TransitionType = ‘Fork’

1.5.1.6 else if (ParCntIncoming > 1) AND (ParCntOutgoing > 1) then

1.5.1.6.1 TransitionType = ‘Complex’

1.5.1.7 Update all codelines in IncomingStateList’s field, Transitioned, to 1 to remove them from all future consideration

1.5.1.8 Create Transition(TransitionType, IncomingList, OutgoingList)

1.5.1.9 Update CurState to first codeline of Outgoing List

1.5.2 if a branch code // denoted by the presence of a populated BranchOutCodeline, Condition, and FirstBlockCodeLine in the CurState

1.5.2.1 TransitionType = Simple // no parallelism in a control construct

1.5.2.2 IncomingList <- codelines in procedure with same sequence number of CurState (PrevStateSeqNo)

1.5.2.3 OutgoingList <- codeline where FirstBlockCodeLine = codeline's codelineID

1.5.2.4 Update all codelines in IncomingStateList's field, Transitioned, to 1 to remove them from all future consideration

1.5.2.5 Search for first codeline after previous codeline (CurState (PrevStateSeqNo) containing an if/while condition) with the if/while level and sequence number within the same procedure. This first codeline becomes the FirstBlockCodeLine (first codeline in control block). If not found, FirstBlockCodeLine is set to end transition's codelineID

1.5.2.6 OutgoingList <- codeline where FirstBlockCodeLine = codeline's codelineID

1.5.2.7 Create Transition(TransitionType, Condition, IncomingList, OutgoingList) // Create Branch-In Transition

1.5.2.8 Search for first codeline, after the previous codeline, outside condition block of previous codeline [CurState(PrevStateSeqNo)] containing an if/while condition) with the if/while level minus 1 (indicating control block has end) and same sequence number within the same procedure. This first codeline becomes the BranchOutCodeLine.

1.5.2.9 OutgoingList <- codeline where BranchOutCodeLine = BranchOutCodeline's CodeLineID

1.5.2.10 Create Transition(TransitionType, NOT(Condition), IncomingList, OutgoingList) // Create Branch-Out Transition

1.5.2.11 Update CurState to first codeline of Outgoing List

1.5.4 CurStatePrevious <- CurState

1.5.5 CurState <- first codeline of outgoing transition list

1.5.6 CurStateSeqNo <- sequence of of first codeline of outgoing transition list of the procedure

8.5 Sample of WSL Code to Activity Diagram

A small sample of WSL code is presented with a corresponding UML activity diagram based on this code.

WSL Code Sample:

```
X := Y + 1
If X > 4 Then
    Fetch D1, Y2, Z2
Else
```

```
        Call B.UpdateRec(X)
    Fi
    J := D1 + X
    M := N + 2 /* potentially parallel operation */
    K := 3 /* potentially parallel operation */
```

The following diagram models the WSL code sample as action states in an activity diagram. Each action state is labeled by the WSL statement whose entry action the state represents. Each action state is placed in the object swimlane whose object produces the action. An example, the invocation of Class B's UpdateRec method is represented as an action state in the Object B swimlane. Potential parallel operations, such as "M := N + 2" and "K:= 3", are modelled as parallel flows emanating from a fork synchronisation bar. An if-then-else WSL construct is modelled in the activity diagram as a branch, with mutually exclusive guard conditions, to two action states. Using these guard conditions, the control flow in the activity diagram is governed in a similar manner to the system that it represents. Potentially parallel executing WSL code lines are modelled as parallel control flows in the activity diagram.

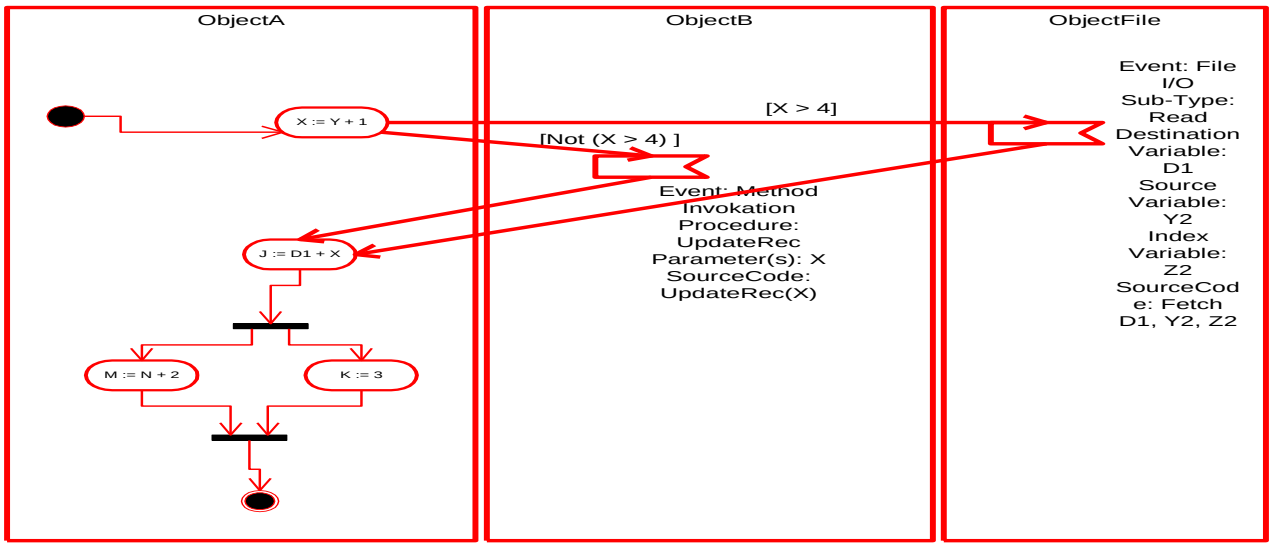


Fig. 8.5.1 : Activity Diagram Representation of the WSL Code Sample.

8.6 Constraints of Specified Activity Diagram Extraction Method

Activity diagrams are useful in modelling the workflow processes of a system. Because the selected unit of granularity for activity diagrams, individual codelines, is very fine, one could argue that these diagrams are cluttered with too many details. Activities should represent procedures or program blocks rather than individual codelines.

However, this argument overlooks several facets. One facet is procedures within this selected legacy system are quite small (Millham, 2003) and that procedures are already represented as entities within sequence diagrams. Furthermore, basing guards on individual codelines is useful in many procedures of this selected legacy system because the code block enclosed by the condition

consists of a single or a few codelines (Millham, 2003). Consequently, basing the activity on the code block enclosed by the condition construct is very similar to that of an individual codeline granularity. In the selected sample system, program blocks with many codelines occur usually during a record to record input/output operation. However, rather than model each individual record field of each record being updated as a separate activity, the generated activity diagram simply models the update on a record level, rather than record field level, basis.

Because batch-oriented systems, such as the selected legacy system, do not have events other than the arrival of input and the departure of output data, modelling of events within activity diagrams must be derived from within the processing of the batch-oriented system. In this case, because the only source of reengineering information is source code, the events identified are pseudo-events (nodes of a control graph constructed of the processing that occurs in this batch-oriented system) and thus, are modelled as such in the activity diagram.

If a different type of system other than a batch-oriented system was used, such as a highly-reactive system used in nuclear fission control, a stimulus-system response graph recording and analysis could be used in order to determine the possible event and system responses. In this manner, a statechart of external events (stimulus) and states (system responses) could be formulated. However, this stimulus-system response system is dependent on many factors such as ensuring that all possible events that the system may face are represented as stimuli. This comprehensive set of stimuli is difficult to obtain and, in the problem domain of many system, constantly changing.

8.7 Summary

The algorithms presented in this chapter serve to simplify and model the control and data flow, as represented by the WSL intermediate representation of the source code, through UML activity diagrams. The main purpose of this visualization, through UML activity diagrams, is to enable the developer to more easily discern the control logic embedded within the program and to understand how data is manipulated within the program. By utilising pattern matching on file operations, many redundant activities, which represent a single record operation, are simplified into a single activity for easier readability. By splitting the activity diagrams into procedures and packages, the original logical cohesion of the program is retained.

Chapter 9: Deployment Diagrams and Component Diagrams

9.1 Introduction

In this chapter, the different viewpoints of the system are listed with a focus on the architectural view, as represented by the component and deployment diagrams. A method to extract component and deployment diagrams, indicating the inter-relationships of program files and the presence of identified physical devices respectively, is provided.

9.2 Different Viewpoints of the System

The inclusion and exclusion of UML diagrams given the sample system was designed to meet the following criteria:

- the static structure of the system, best expressed via a class diagram
- the behavioural aspects of the system, best expressed via activity diagrams
- the dynamic aspects of the system (how objects interact with one another in the system) as expressed in sequence diagrams
- the architectural view of the system (how different subsystems are composed within the larger system), as expressed in component diagrams and in how different peripherals of the system are connected to the system, as expressed in the deployment diagram.

9.3 The Architectural Viewpoint and the Selected Sample Legacy System

Five of the possible nine UML diagrams are available through the TAGDUR tool. However, the architectural view of the system is not as straightforward as in other batch-oriented systems. The selected system forms a standalone, core business system of a telecommunications company. When it was designed and developed during the 1970's, the original user interface consisted of punched cards with the occasional dumb terminal. Output was also provided via punched cards, tape, printers, and terminals. However, as time went on, this user interface architecture proved unwieldy and unsuited to the emerging adopted architecture of the corporation. Rather than rely on dumb terminals or punched cards for system input and output, screen scrappers, usually written in Visual Basic and available on users computers, scrapped the user-inputted information from the client screen, packaged it into a request file, and then put it in a specified network location. The core system would then periodically poll this specified network location for the presence of these input files and if found, unpackage them and input them into the system, as part of the input file, in the same manner as the old punched cards. When this system had finished processing this input file, it would then repackage the processed output into an output file package and then put it into another specified network location. The client's screen scrapper application would periodically poll this specified output network location for the presence of this output file.

Once detected, the client application would retrieve the output file, unpackage it, and then display it to the user for further processing (Telecommunication Employee C).

This business process had several implications. One, the original user interface of terminal, printer, and other peripherals have been removed. Input and output are now exclusively handled by files. Hence, a reengineering tool's method of parsing the source code for the presence of these peripherals and the links that these peripherals have with their enclosing procedures, classes, packages, et al would be suitable for systems with this type of interface but not with this sample system. Secondly, this process of polling and file handling is an extremely slow one. It may take several hours for the system to process the input file (a request) and return an output file (a reply). This time lag is highly unsuited for quick customer service and for a Web-based, customer driven interface. What customer would like to put in a customer service order on the Web only to be told to check back several hours later to see if it is possible? Hence, the tremendous need to reengineer this system, if only for speed (Telecommunication Employee A; Telecommunication Employee C).

9.4 TAGDUR's Ability to Produce Deployment/Component Diagrams vs. Selected Sample System

Thus, the ability of the thesis tool, TAGDUR, to produce component diagrams of software components as part of this system is limited by this sample system. In many systems, parts of the system, in this case COBOL copybooks, are incorporated into other parts of the system through the COPY keyword which simply copies the sub-file into the calling file in much the same way that a C program can incorporate a library of functions into itself by specifying this library in its header section. In the TAGDUR tool, the source code is parsed for the INCLUDE command to indicate what sub-files should be included in the source file being parsed and to depict this inter-relationship via a component diagram. These inter-relationships of various sub-files are important in any large system with many sub-files. However, in this selected system, parts of the system, such as individual COBOL files, are called via JCL (the Job Control Language) which is outside the COBOL program. Hence, parsing the COBOL source code to find which sub-file is included in which program file and consequently extracting a component diagram, while theoretically possible to perform, is not feasible for this particular legacy system.

9.5 Component Diagram Extraction

Component diagrams depict the run-time relationships among software components of a system. These components may be simple files or libraries loaded dynamically. The relationships among software components are usually compilation dependencies (Muller, 2000a: pp 133-134). Deployment diagrams depict the physical arrangement of various hardware components and executable files of the system (Muller, 2000a: p 136).

The Component UML diagram extraction method consists of parsing the source code in order to find the keyword, **include**, in the following format **include** <*LibraryName*> in the source code file, <*SourceCodeFile*>. that is being parsed The library name is extracted and a relationship between the software component, <*SourceCodeFile*> , and the <*LibraryName*> is depicted as a dependency relationship between two software components in the extracted UML component diagram.

9.5.1 Algorithm to Extract Component Diagrams

To extract the component diagrams from the WSL code, dependencies between software components of a system need to be identified. The algorithm for this extraction is as follows:

1. For each WSL source file <SourceFile> in the legacy system

1.1 Parse the source code in the file, <SourceFile>, for specific keywords that identify an external component and the resulting compilation dependency between this component and the source file

1.1.1. If the statement **Include** <FileName> is encountered, establish a compilation dependency link, if one does not already exist, between the <FileName> and <SourceFile>

1.1.2. If the statement <FileName> **.DATA_RECORD :=** <FileRecord> is encountered, establish a dependency link, if one does not exist between the file record, <FileRecord>, and its associated logical file name, <FileName>.

9.5.2 Example of an Extracted Component Diagram

If source code file SCF2 contains a WSL statement that loads a library file, LF2, this loading is depicted in the component diagram as a dependency relationship between two software components SCF2 and LF2. In order to represent this dependency relationship between a software component and an associated library file, a new WSL COBOL construct is introduced in the format **include** <*LibraryName*> where <*LibraryName*> represents the library file name which is included, via a dependency relationship, into the calling software component. The **include** WSL construct represents the COBOL construct, *COPY <CopyBookName>,* where <*CopyBookName*> is the name of the library file that is copied into the calling software component during compilation time (Brown, 1999).

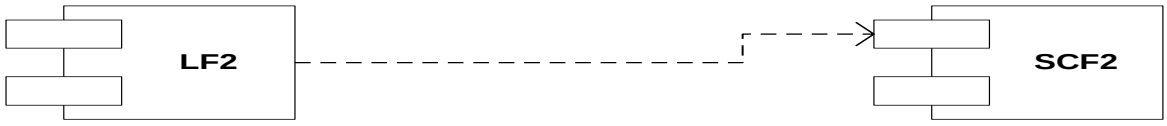


Fig.9.5.2.1 Component Diagram Showing Compilation Dependency between library file, LF2, and source code file, SCF2.

9.6 Extraction of Deployment Diagrams

To obtain the relationship between source code files and physical devices, the source code is parsed for keywords denoting physical devices. The WSL statement, **Put IOTerminal**, X, 1, in program file P1 indicates a relationship between file, P1, and

peripheral device, IOTerminal. To extract the deployment diagram that shows the relationship between physical files and source code files, the source code is parsed for the WSL keyword, **Phys_Dev** in the format **Phys_Dev** := <PhysicalDeviceName>. The presence of this keyword with the <PhysicalDeviceName> shows a relationship, depicted in a component diagram, between <PhysicalDeviceName> and the source file being parsed. Furthermore, each physical file device has a supertype, File, which is a pre-defined physical node that represents the generic file system of a platform.

9.6.1 Algorithm to Extract Deployment Diagrams

In order to extract deployment diagrams from the WSL code, the WSL code files need to be parsed to identify the physical devices called from within code files. The algorithm for extraction of deployment diagrams is as follows:

1. For each WSL source code file <SourceCodeFile>.

1.1 Parse the source code in the source file, <SourceCodeFile>.

1.1.1 If the keyword, IOTerminal, is encountered, establish a link, if one does not already exist, between the <SourceCodeFile> and the peripheral device, Terminal.

1.1.2 If the keyword, IOPrinter, is encountered, establish a link, if one does not already exist between the <FileName> and the peripheral device, Printer.

1.1.3 If the keyword, IOKeyboard, is encountered, establish a link, if one does not already exist between the <FileName> and the peripheral device, Keyboard.

1.1.4 If **LOGICALFILENAME.PHYS_DEV :=** <PHYSICALDEVICE> is encountered, establish a link, if one does not already exist between <LogicalFileName> and the physical file device <PhysicalDevice>. Another link should be established, if one does not exist, between the physical file device and the abstract device, File, which represents the system's entire file/database system.

9.6.2 Example of a Deployment Diagram

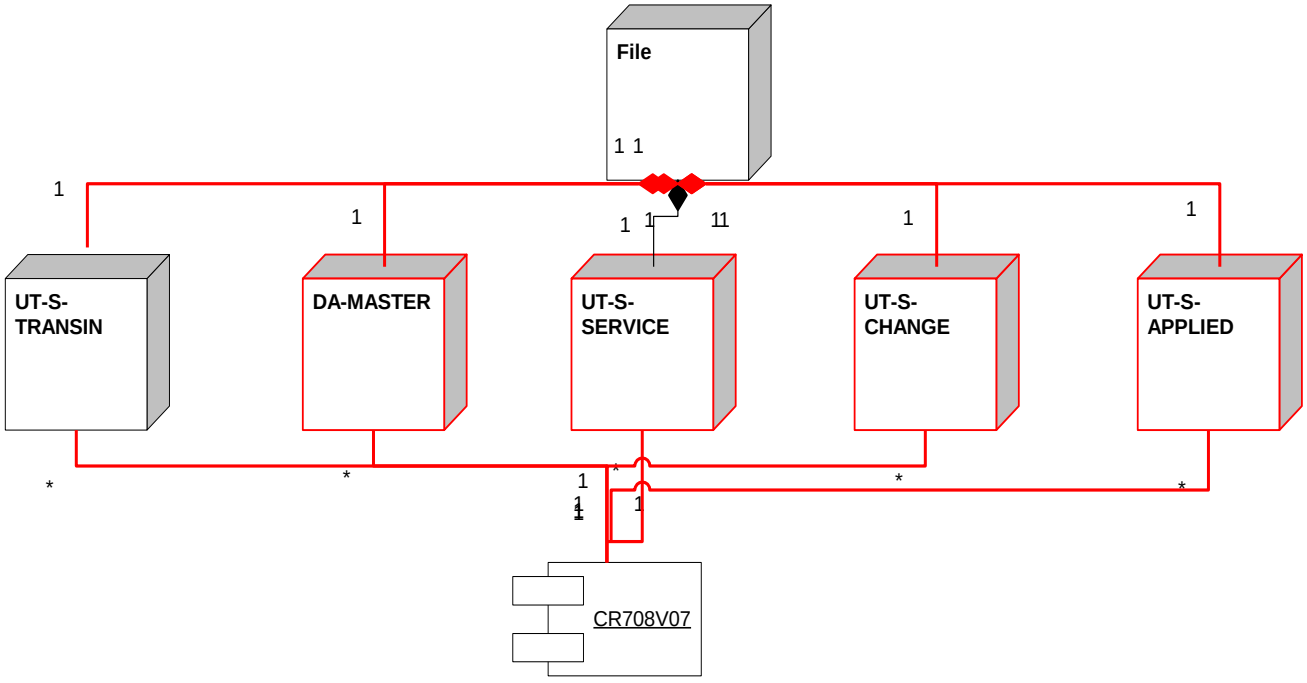


Fig.9.6.2.1 A deployment diagram indicating the physical device mapping to physical file system to physical COBOL source file, CR708V07.

Figure 9.5.1 is an example of the mapping of logical physical device names to a software component, CR708V07, of the selected legacy system. It is important to depict these relationships in a component diagram in order the dependencies of software components and the physical devices used by the legacy system platform are known, especially if these physical devices are moved or replaced.

9.7 Summary

Legacy systems often are composed of many inter-linked source code sub-files. This chapter provides a method to extract from the source code, these relationships, and to depict them in a component diagram. A method is also provided to extract the deployment diagrams from source code by identifying, through the parsing of program files, the presence of physical devices and their relationship to source files. However, due to the removal of the original COBOL system interface of punched cards and terminals, the physical file relationships to program files are the only remaining artefact as depicted in the component diagram.

Chapter 10: Use Case Diagrams, Collaboration Diagrams, State Charts and Object Diagram Extraction

10.1 Introduction

In this chapter, the remaining UML diagrams (collaboration, object, state chart, and use case) are outlined and reasons why these diagrams could not be satisfactorily extracted from the source code of this type of legacy system are provided.

10.2 Collaboration Diagrams

Because collaboration diagrams are very similar to sequence diagrams with some relation to class diagrams in that emphasise the structure and relations of objects within the sequence of events (Oestereich, 1999), it was considered redundant to extract a very similar UML diagram, collaboration, from the source code when a UML sequence diagram (see algorithms outlined in Chapter 7) and a UML class diagram (see algorithms outline in Chapter 6) has already been extracted.

10.3 Object Diagrams

In a similar way, since there is exactly one object per class in the thesis' representation (as outputted by the algorithms outlined in Chapter 6), there is no need to represent objects separately. If there were multiple instances (objects) of the same class, an object diagram would be needed to represent what instance of a particular class is present. Additionally, object diagrams often have a lifetime where they show when a particular object is created and when it is destroyed. Since the TAGDUR tool lacks a sophisticated garbage tool collection facility, objects are created at the start of the application and destroyed at the application's termination. In the future, it is hoped that a more sophisticated approach can be implemented where class objects are created when needed and then destroyed when they are no longer in use.

10.4 Statecharts

Statecharts, because they show individual states along with pre-conditions and post-conditions of each state, are best suited for forward engineering of highly-reactive systems. Deriving a properly-formulated statechart from source code is very difficult because it is impossible to determine how a system would react in a given situation. An example, one could derive a list of pseudo-states from operations inherent in source code but these pseudo-states do not take into account the underlying architecture or program. An example, if the source code specifies an arithmetic operation and an overflow occurs, how can this overflow be predicted and accurately modelled? Systa has proposed a method of deriving statecharts by the use of a trace analysis of the system response given certain stimuli (Systa, 1997). An example, suppose one inputs a typical set of data into the system, the system's responses to this data stimuli are then measured and using the stimuli-response trace analysis, a chart of states with

pre and post conditions can be formulated. Systa's method for this selected sample system is impractical for several reasons. The typical set of data for this system is unavailable for several reasons. Due to confidentiality reasons, the original telecommunications company was unwilling to release a set of typical user requests which could be used to provide a set of stimuli. Another reason is that often the typical set of data changes over time; an example, during the 1970's, most user requests were for installations of basic telephone equipment while in the 2000's, an increasing set of user requests are for wireless and data services. Because this system can handle these different types of requests quite differently, it is very difficult to model a statechart based on a constantly changing set of data.

Systa's trace analysis of stimulus-response by the system has additional drawbacks in modelling statecharts. Even with trace analysis, it is very difficult to determine the exact sequence of actions taken by the system in response to an event stimulus and hence, it is difficult to model the exact set of states and transitions, in a statechart, that a system would proceed through in response to a particular stimulus. This trace analysis is highly dependent not only on the software system being analysed but on the underlying operating system and the hardware platform. An example, given a mouse click stimulus and a mouse click event handler in the software system, how does the underlying operating system handle such an event? Some operating systems continuously poll for such events while others generate an interrupt. Even if an interrupt is generated, how is the interrupt handled, particularly if the software system is busy handling a previous interrupt? Some operating systems will disable the new interrupt until the previous interrupt has been fully handled (Tannenbaum, 1992). Statecharts would have to model the sequence of actions taken by the system until the event stimulus is detected by the operating system and then handled by the software system's event handler. Depending on how the operating system, as one example, handles event stimulus and how it handles multiple simultaneous interrupts, the statecharts depicting the same set of event stimulus using the same software system would be quite different depending on the operating system being used.

10.5 Use Case Diagrams

Due to the great age of this selected sample system and the lack of documentation, coupled with the large number of personnel, who originally had knowledge of this system and who have subsequently left this organisation (the original user/developer base) upon which use cases could be built, extraction of use cases in order to model the system's business processes is no longer feasible. In the absence of information, whether documentation or the user/developer community that can provide accurate and complete information on the real world environment upon which the software system models, the development of complete and accurate use cases is a very difficult task. Information regarding this real world environment from source code alone is impossible. Furthermore, the development of use cases requires a large amount of user interaction and the design of this tool was to automate this process of reengineering as much as possible with minimum user interaction. However, the parsing of the source code can produce a large volume of information, such as conditions under which information is manipulated, from which a number of business rules can be extracted (Aiken, 2000). The extraction of business rules from the result of the parsing of the TAGDUR tool is a future development.

10.6 Summary

This chapter outlines which of the nine UML diagrams can not be extracted satisfactorily from a batch-oriented system. Some diagrams, such as collaboration or object diagrams, were not extracted because of their similarity to the different extracted UML diagrams of sequence and class respectively. The one class to one object correspondence in the class diagram with little hierarchy of classes, along with the system execution time lifetime of objects make class diagrams very similar to object diagrams and

consequently, make object diagram extraction redundant. Because this system is batch-oriented with few, if any, external events, the system's dynamic behavior can be better expressed through activity diagrams rather than statecharts, which are better suited in modeling the behavior of highly-reactive systems with many diverse external events. Relying on system source code as the only system artefact and without significant human input or system documentation, the derivation of meaningful use case diagrams from this legacy system, under such constraints, can not be accomplished satisfactorily.

Chapter 11 : Tool Design and Experiments

11.1 Introduction

In this chapter, TAGDUR's architecture and the rationale for its design is described. A description of the tool's operation is provided. Four samples of WSL code, of varying sizes and containing various attributes, are parsed, analysed, and their output, in the form of WSL-UML notation, is displayed in statistical form. One sample is also displayed in UML diagrams in Appendix C.

11.2 Tool Design and its Rationale

The thesis' tool, TAGDUR (**T**ransformation and **A**utomatic **G**eneration of **D**ocumentation in **U**ML through **R**eengineering), was designed to try and overcome the lack of documentation problem often faced by legacy systems whose original documentation has been lost. By utilising information acquired during the transformational process and by parsing the code of the transformed system, this tool generates UML diagrams of the transformed system.

11.3 Tool Design and Experiments

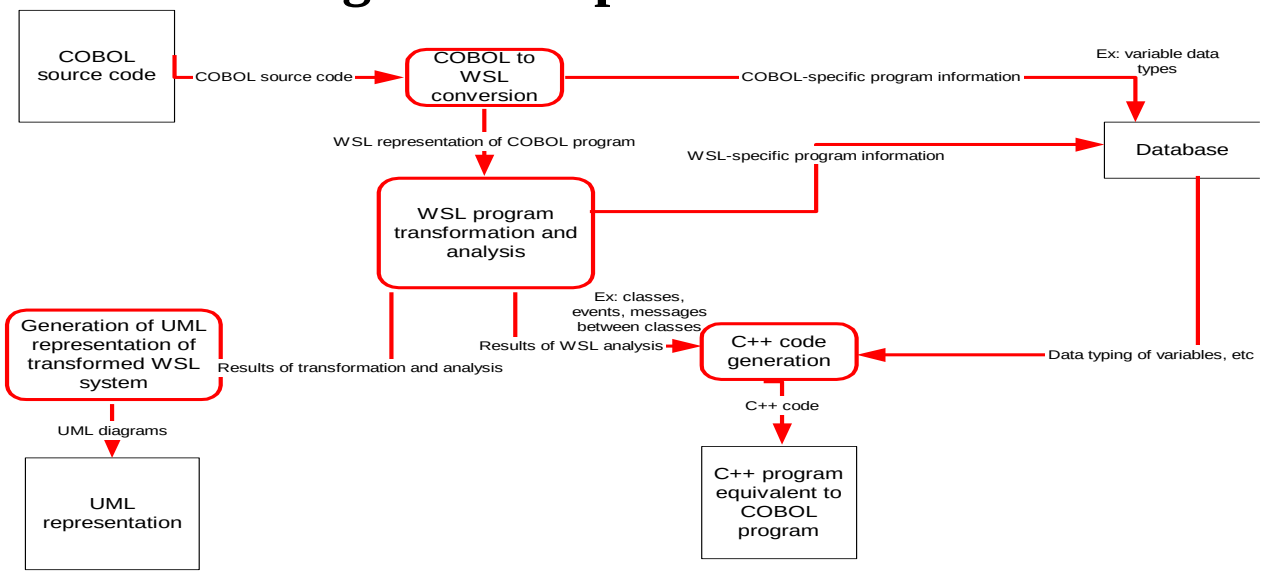


Fig.11.3.1 Overview of TAGDUR Tool Design.

11.3.1 Basic Architecture

This tool consists of five basic components:

1) A series of COBOL source code files in textual format

2) A series of manually converted WSL representations of these COBOL source files, in textual format

3) A parsing, reengineering tool written in Visual Basic 6

3) A database, originally Microsoft Access and now SQL Server 2000

4) A series of files, representing a series of UML diagrams of the selected system, in textual format

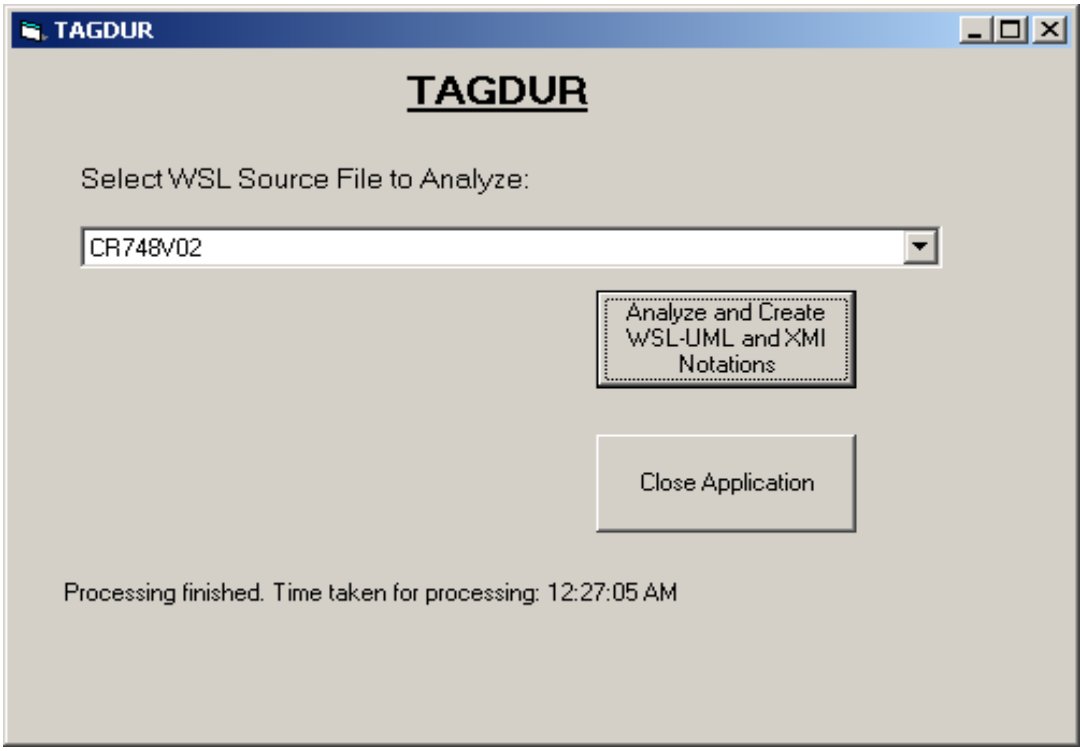


Fig 11.3.1.1 Main File Selection for Analysis and Extraction Screen.

It was decided to represent the COBOL, WSL, and UML notations in textual format for the reasons of easy parsing, viewing, and compactness. By placing these types of files in small text files, they can easily be transferred in and out of the reengineering tool's input/output queue. The source files, whether COBOL or WSL, are placed in specified file locations to be opened and parsed by this reengineering tool; the output files are placed in specified file locations to be picked up and viewed by the user. The users are able to select, from a list of WSL files, which file they wanted parsed, analysed, and converted to XMI format to form UML diagrams. Once selected, the users may click on the "Analyze and Create WSL-UML and XMI Notation" button to produce this XMI format. The user can then open a UML diagramming tool, such as Poseidon, and import the generated XMI format to produce UML diagrams. As the tool proceeds with the restructuring and UML extraction processes, the user are informed of the exact stage in this reengineering effort through messages displayed on screen

A database is required because the information obtained during the reengineering effort can be quite huge and a database, by its nature, is well suited to handle this volume of data. As well, the data, as existing in the database, can easily be selected and manipulated through the use of database queries.

Our parsing tool was written in Visual Basic 6 due to its interactive development environment and its availability to run on its original platform, a Windows-based PC. Although the Visual Basic version 6 language might seem unsuited to the development requirements of an industrial strength reengineering tool, the algorithms developed and used within this tool can be transferred into most imperative programming languages.

11.3.2 Tool Operation

The COBOL legacy system was first converted into WSL using a set of COBOL to WSL conversion rules. These rules were formulated using Martin Ward's (Ward, 1992) paper, *The Syntax and Semantics of the Wide Spectrum Language,* which defined the basis of the Wide Spectrum Language. Because WSL is a type less language, programming language-specific information, such as variable data types of the original legacy system, can not be represented in WSL but are stored in the database for future use, such as during the C++ code generation process.

Because of the difficulties with automating COBOL to WSL conversion, this conversion was accomplished manually using a set of rules defined in this thesis (see Chapter 5).

After manual conversion of COBOL source files to WSL has been completed, the parsing component of the tool parses the WSL-equivalent source files and during this parsing, the program within the source file is syntactically analysed for variable declarations and usage, procedure declaration and usage, control constructs, etc. This information is stored in the database.

After the parsing and syntactic analysis have been completed, TAGDUR begins its transformation steps to restructure the original procedural-structured and sequential legacy system to an object-oriented, event-driven system. This transformation occurs using a series of restructuring steps. The first step is to identify potential classes, including their attributes and operations through the thesis' own clustering technique. This technique groups highly inter-related procedures and variables into classes (Newcomb, 1995). The next transformation step is to evaluate tasks, at both the procedure and individual code line level of granularity, for their degree of task independence. This evaluation is accomplished using several different algorithms as outlined in (Millham, 2003a). A task is defined as an atomic unit of work; tasks can be defined at several different levels of granularity such as program, procedure, and individual code line. Our task evaluation technique involves analysing two normally sequential tasks for data and control dependencies between them; if a dependency exists, these tasks are deemed to be sequential; if no dependency exists, these tasks are deemed to be able to execute in parallel. The final transformation step is to identify possible events from source code. This event identification step is accomplished by constructing a control graph of procedure calls, I/O calls, system interrupts, and error invocations by parsing the source code. The nodes of this control graph that represent interaction with other objects are modelled as events. An example, an interrupt is modelled as an event occurrence between the object where the interrupt occurred and the System object, which handles interrupts. Once the events are identified, each event is evaluated in terms of its synchronicity. This evaluation is accomplished by evaluating, at the individual code line level of granularity, the task where the event occurred and the task immediately successive to this task in order to determine if any control or data dependencies exist among them. If a dependency exists, the event is deemed to be synchronous; otherwise, if no dependency exists, the event is deemed to be asynchronous. The information that was gained during these transformation processes is stored in the database.

The information that was obtained during the transformation process is utilised during the extraction of UML diagrams process of TAGDUR. An example, classes identified during the class identification process are modelled as interacting objects in the sequence diagram. Events identified during the event identification process are modelled as messages, either synchronous or asynchronous, between the sending and receiving objects. The independent task evaluation process determines the ordering of

tasks where each task is given a sequence number, in ascending order of execution. If two tasks may execute in parallel, the two tasks are given the same sequence number. The sequence number of the task enclosing the event, such as a procedure call, is used as the sequence number of the message in the sequence diagram.

Once the transformation processes are complete, TAGDUR then begins its processes of extracting a series of UML diagrams, where feasible, from the legacy system. These extracted diagrams are then represented in WSL-UML notation, exported to a XMI exchange format, and then imported into a visual UML modelling tool.

11.4 Rationale Behind Tool Design Decisions

Existing WSL tools, such as Maintainers Assistant, were not used for several reasons. One reason was that none of the existing tools provided restructuring from a procedural to an object-oriented system. Another reason was that none of the existing WSL tools produced UML diagrams from their analysis of the source code.

UML diagrams were selected as the method to document the transformed system for several reasons. UML provides documentation of the system from multiple perspectives; UML provides use-case diagrams, which address the modeling needs of end-users, but also provides statecharts and class diagrams, which are most useful to developers. The diagrams that form UML, diagrams such as statecharts, have been used by the software community for years to model and to understand software systems. UML is an accepted worldwide standard with significant tool support. Furthermore, UML is independent of any programming language and platform. Although use case diagram generation is not a current feature of TAGDUR, TAGDUR embraces class and activity, which are similar to statechart, diagrams in its UML modeling.

WSL was chosen for an intermediate language for several reasons. WSL is programming and platform independent. Consequently, the transformations and modelling that TAGDUR performs on a WSL-represented system could be performed regardless of whether the original legacy system was in COBOL or C. The original legacy systems need only to be converted into WSL first.

WSL has other advantages as well. WSL has excellent tool support, in the FermaT transformation system (Ward, 2001) which allows transformations and code simplification to be carried out automatically. WSL is programming and platform independent. Because WSL can represent both low level and high level views of a system, as per its wide spectrum nature, it is ideal for analysing and modelling low-level source code in high level modelling diagrams.

Basing the thesis' tool on using the only surviving system artefact, source code, of many legacy systems, it was decided that the TAGDUR tool would be a parsing tool that parsed the source code in order to obtain relevant information about the system and then model this information through a selected series of UML diagrams.

11.5 Advantages of TAGDUR and its Comparison with Other Reengineering Tools

Rumbaugh et al identifies three viewpoints necessary for understanding a software systems: the objects manipulated by the computation (the data model), the manipulations themselves (functional model), and how the manipulations are organised and synchronised (the dynamic model) (Rumbaugh, 1991). In this thesis, another view is proposed as being required for system understanding, the architectural view, which represents the physical components of the system and the relationships among them.

TAGDUR addresses all four of these views through its generation of UML diagrams: the static, behavioral, dynamic, and architectural views through the class, activity, sequence, and deployment or component diagrams respectively.

Sequence diagrams, which represent the dynamic view, are useful in depicting the messages passed between interacting objects. The developer needs to understand the interactions among objects as depicted in a sequence diagram in order to properly design the interfaces of these objects. If the developers plan to distribute objects among several different platforms, such as in a multi-tiered Web architecture, the developer would need to understand the object interactions in order to group frequently-interacting objects on the same platform in order to minimise communication costs (Ulrich, 2000).

The architectural view is required in order to understand how the system is related to external physical entities, such as a printer, and the dynamic configuration of program files (Mansurov, 2003). This architectural view is need if a physical entity, such as a database file, changes and the developer needs to quickly ascertain which program files access this database file in order to change the relevant code within them. In a large legacy system, such as the particular telecommunications legacy system used in the thesis' study, there are 105 source code files. It is necessary to depict the relationships among source code files, as in which source code files load that other source code files as libraries, in a component diagram because the number of potentially loadable libraries is often too numerous to keep track of manually.

These four views provided by TAGDUR differentiate it with other tools. The Klockwork tool extracts objects, packages, and components (Mansurov, 2003). However, it does not restructure existing code from procedural to an object-oriented paradigm nor does it provide a dynamic view, unlike TAGDUR. Similarly, no restructuring is provided with Harris' framework (Harris, 1995), Murphy's tool (Murphy, 1996), Tonnella' method (Tonnella, 2003), RAISE (Univan, 2000), and the Moose and Embercardo tools (Embercardo, 2004; Bertouli, 2003). TAGDUR provides a practical tool rather than a methodology, unlike those of Jacobson and Lindstrom (Jacobson, 1991) and Liu and Wilde (Liu, 1990).

Several existing tools provide restructuring from procedural to object-oriented systems. Zhou restructures a procedural to an object oriented system using persistent data stores and related function calls; since an intermediate language representation, such as WSL, is not used, the objects identified are highly language dependent. Furthermore, unlike TAGDUR, only class diagrams are extracted, rather than a more comprehensive view of the system using activity, component, and sequence diagrams (Zhou, 2003). Many of these restructuring tools require expert user validation for their restruturing, either in full or in part, such as RIGI (Storey,

1995) or Moore (Fergan, 1994). Furthermore, RIGI's and Moore's re-documentation is not through an industry standard modelling notation like UML but through proprietary call graphs and other methods of documentation, which developers may find difficult to understand. Furthermore, inter-modular procedural calls and calls to external procedures are not modelled, unlike TAGDUR's extracted activity and sequence diagrams (Wong, 1995).

The Refronte tool restructures and re-documents various types of source systems but it, unlike TAGDUR, it does not provide a dynamic or behavioral view of the system (through sequence and activity diagrams) but provides only classes and component identifications (Jarzabek, 1995). Eqyed's tool focused on dynamic system modelling using a scenario tracing approach to produce a dynamic behavior view of the system yet no restructuring was provided and the documentation was a mix of object-oriented, in the form of class diagrams, and functional, in the form of dataflow diagrams, styles (Egyed, 2003). TAGDUR, although it does not provide a dynamic, but rather a static, behavior view, provides documentation in an object-oriented style, through UML diagrams, only.

Van den Brand's tool focuses on restructuring COBOL system using a formal intermediate language representation, this tool focuses on one dialect of COBOL with many of their production rules left out producing an incomplete translation (van den Brand, 1997c). Although TAGDUR uses a COBOL legacy system, it can also represent other types of source systems, such as C or C++. Sneed restructures COBOL programs from procedural to object oriented systems but the resulting objects are non-standard (Sneed, 1988). TAGDUR provides standard objects from its extraction and models them in class diagrams.

Neither Fermat nor Maintainer's Assistant are able to handle COBOL source code systems whose source code is COBOL nor do they provide an abstract model of the system in the form of UML diagrams (Ward, 1989; Ward, 2001).

TAGDUR addresses a gap that is found in existing reengineering tools. First, a practical tool is provided rather than just a reengineering methodology. Secondly, it converts the original source code into a formal intermediate language representation that enables the restructuring and UML extraction algorithms to be independent of the source programming language and, thus, consistent in its object clustering and other processes. A formal intermediate representation has additional advantages such as its preciseness. Thirdly, it restructures a procedural into an object-oriented, event-driven system using a series of validated restructuring algorithms. Fourthly, using a static analysis of code combined with information of the system gained during the restructuring phases, a systematic set of documentation is extracted from the intermediate language representation and provided via a series of UML diagrams (class, activity, sequence, and component), that have been validated through a UML checklist (see Appendix H) to ensure diagram completeness and representational accuracy, to represent the structural, behavioural, dynamic, and architectural views of the system respectively. Furthermore, the formal intermediate representation, WSL, was extended to represent abstract modelling notations through the WSL-UML notation. This tool addresses a gap in current reengineering tools in that it both restructures a procedural legacy system to an object oriented system without expert user guidance, and the re-documents the full view of the system (defined as encompassing the structural, behavioural, dynamic, and architectural views of the system) using a standardised systematic form of documentation.

The TAGDUR tool has additional features as well. The tool has the nascent capability of generating code for a C++ equivalent program of the transformed system (see Appendix D for some general rules regarding WSL to C++ conversion). Future

additions to this tool will include the ability to data reengineer the underlying databases or file systems of the system. Another future addition will be the ability to generate test scripts and cases from activity diagrams; these test cases provide a means of regression testing the various iterations of the transformed system as it is being redeveloped in order to integrate this system with other related systems.

11.6 Advantages of TAGDUR in Redeploying to Web-based Platforms

One of the advantages that TAGDUR possesses in restructuring legacy systems is that it better enables a legacy system to fit into a Web based environment. One of the proposed goals of the selected legacy system was to move it to a Web-based platform in order to enable clients to interact with the system directly rather than through an employee, with the consequent cost savings of such a redeployment (Telecommunications Employee C, 1999). One common solution to this redeployment is to use an object wrapper for the legacy system; this object wrapper, however, requires first a strong knowledge of the legacy system, which may be missing, and encompasses many of the problems that object wrapping involves such as static functionality and high maintenance costs (Bisbal, 1999). Moving from a mainframe legacy system to a Web platform usually involves a restructuring of the system from a procedural to an object-oriented and event driven system (Harris, 1995; Zhou, 2003; Ulrich, 2000).

Object orientation, which the restructuring feature of TAGDUR enables, encapsulates methods and their variables into modules that are well-suited to a Web-based environment where pieces of software must be defined in encapsulated modules with clearly-defined interfaces (Newcomb, 2001; Ulrich, 2000; Zhou, 2003). Another advantage of object orientation is component reuse. A class or object of a legacy system may be taken out, modified, tested, and then replaced with little overhead and downtime. There is also less risk of the changes rippling through the rest of the system in unintended and unpredicatable ways. Another advantage is TAGDUR's extracted sequence and potential deployment diagrams. When the new objects of the legacy system are scattered over several machines, as in a Web farm, knowledge of object interaction, as depicted in the sequence diagram, is key in order to prevent unwanted side-effects as classes interact from one machine to another. Knowledge of peripheral devices and the objects that they interact with, available through deployment diagrams, enable the developer to locate those objects that frequently interact with peripheral devices in an optimal way in order to reduce network traffic (Gunderloy, 2003; Wijegunarante, 1998).

11.7 Description of Experiments on Selected Samples of System Code

11.7.1 Selection of Code Samples

The original selected legacy system used in this thesis consists of approximately 112 000 COBOL lines of code in 105 different sub-files or modules. Because of the huge size of the system and the need for manual conversion of this COBOL source code into a WSL representation, the full-scale conversion of such a large system is a monumental task. In order to demonstrate the ability of the thesis' tool, TAGDUR, to handle the restructuring and re-documentation of a system, several representative COBOL sub-files of this selected legacy system were chosen. These selected samples varied in size: one was a small sample, two were medium sized samples, and the last one was a large sample. These samples also varied in code composition, functionality, and in programming style. Different sub-files were written by different programmers at different times. One sample consisted of many variable declarations with relatively little procedural code while another sample consisted of mostly procedural code with

fewer variable declarations. Other samples contained a more even mix of procedural code and variable declarations. Some samples included intensive control logic with extensive file input-output operations while other samples had little control logic. By selecting samples with varying factors such as sample size, programming style, amount of control logic, and ratio of variable declarations to procedural code, it was thought that these selected samples would be representative not only of components of the selected legacy system but of batch-oriented systems as well. By demonstrating that the thesis' tool, TAGDUR, could handle samples with varying but representative factors, it can be demonstrated that TAGDUR is able to handle, through its algorithms and design, a variety of batch-oriented systems.

11.7.2 Experimental Results of Selected Samples

A set of samples (in their original COBOL, their WSL translation, and their modelling through UML diagrams after restructuring) is given in the Appendices. Because of the large amount of space needed to list all of these samples from their COBOL source code, to their procedural WSL representation, and then to their modelling representation, through a series of UML diagrams, after their restructuring, it was decided to list only one small (CR750V02) sized sample in the Appendices. The sample's, CR750V02, diagrams were extracted and modelled in Appendix C. The remaining samples are summarised in tables below.

The sample, CR750V02, which was selected for displaying UML diagrams, is representative for batch-oriented COBOL systems in that it contains some control logic, some procedure calls, and significant file I/O and assignment operations.

The results of these experiments, using the set of selected samples, is summarised in tables below:

A.CR750V02

Number of COBOL lines	215	
Number of WSL lines	238	
UML-WSL Notation		
UML Component Name	**Sub-Component Type**	**Number of Occurrences**
Classes		6
Procedures/Methods		10
Variables/Attributes		57
Package		1
Comments		26
Associations		10
AssociationEnds		20
Sequence		37
Message		28
	Asynchronous	27
	Synchronous	1
	Containing Guards	7
Partition		27
Deloyment		1
	System Call	0

	Physical-Logical FileName Mapping	1
Activities		10
Action State		30
	Assignment	27
Transition		78
	With Guards	6
Event		37
	Method Invocation	10
	System Interrupt	7
	Error	7
	File I/O	13

Table 11.7.2.1 Results for Sample CR750V02.

B. CR748V02

Number of COBOL lines	540	
Number of WSL lines	749	
	UML-WSL Notation	
UML Component Name	**Sub-Component Type**	**Number of Occurrences**
Classes		10
Procedures/Methods		17
Variables/Attributes		224
Package		1
Comments		42
Associations		51
AssociationEnds		102
Sequence		149
Message		92
	Asynchronous	90
	Synchronous	1
	Containing Guards	11
Partition		26
Deployment		4
	System Call	1
	Physical-Logical FileName Mapping	3
Activities		25
Action State		186
	Assignment	128
Transition		231
	With Guards	22
Event		92
	Method Invocation	25
	System Interrupt	20
	Error	20
	File I/O	47

Table 11.7.2.2 Results for Sample CR748V02.

C. CR702V01

Number of COBOL lines	582	
Number of WSL lines	539	
	UML-WSL Notation	
UML Component Name	**Sub-Component Type**	**Number of Occurrences**
Classes		10
Procedures/Methods		24
Variables/Attributes		74
Package		1

Comments		14
Associations		53
AssociationEnds		106
Sequence		163
Message		56
	Asynchronous	54
	Synchronous	2
	Containing Guards	6
Partition		27
Deployment		1
	System Call	1
	Physical-Logical FileName Mapping	0
Activities		32
Action State		127
	Assignment	113
Transition		188
	With Guards	22
Event		76
	Method Invocation	32
	System Interrupt	20
	Error	20
	File I/O	4

Table 11.7.2.3 Results for Sample CR702V01.

D. CR708V07

Number of COBOL lines	2191	
Number of WSL lines	3790	
UML-WSL Notation		
UML Component Name	**Sub-Component Type**	**Number of Occurrences**
Classes		21
Procedures/Methods		41
Variables/Attributes		855
Package		1
Comments		94
Associations		497
AssociationEnds		994
Sequence		2979
Message		418
	Asynchronous	417
	Synchronous	1
	Containing Guards	118
Partition		27
Deployment		6
	System Call	1
	Physical-Logical FileName Mapping	5
Activities		113
Action State		1038
	Assignment	735
Transition		1044
	With Guards	163
Event		418
	Method Invocation	113
	System Interrupt	42
	Error	42
	File I/O	221

Table 11.7.2.4 Results for Sample CR708V07.

These samples represent a small (215 COBOL lines), two medium (540 and 582 COBOL lines), and one large (2192 COBOL lines) sample. These samples represent common segments of code that would typically found in such a selected legacy system. These samples differ in various ways. An example, sample CR750V02 has tightly-coupled classes with few associations among

classes. It has many method calls encompassed within control constructs but few control constructs outside method calls. It has few assignment calls.

There are differences among the two medium sized samples. Sample CR748V02 has a large number of variables but fewer procedures in proportionate to the other similar-sized sample. Its classes are more loosely coupled with a higher degree of association (inter-object coupling) than its similar size counterpart, CR702V01. CR748V02 has much fewer method calls but more assignment statements than CR702V01. CR748V02 has more control logic controlling its events than CR702V01.

The large sized sample, CR708V07, has a small number of classes relative to its larger number of procedures and variables. It also has more loosely coupled classes with a large number of associations, relative to its size, than CR702V01 but similar, in proportion to its size, with CR748V02. It has the same number of method calls as its much smaller sample, CR702V01. Most of its messages contain control constructs. Most of its action states are assignment states.

In summary, besides the difference in size, these samples differ in the amount of method calls, file operations, degree of control logic, degree of intra and inter class coupling, and composition of action states (assignment versus other statement types). Because of these variances, these samples provide a good representation of the differing types of sub-files that one might encounter in a batch file and demonstrate TAGDUR's ability to handle these differences in its modelling.

11.7.3 Validation of UML Diagrams of Selected Sample

Given a selected sample, CR750V02, a checklist, which was used to ensure well-formedness of the UML diagrams and to ensure the correct correspondence between the WSL code and UML diagrams that were extracted from it, was used to validate that the outputted UML diagrams corresponded to an object-oriented version of the WSL code (see Appendix H). The idea of the checklist was both to formalise and to notate the process that an expert validator, familiar with UML and WSL, would undergo in validating that the generated UML diagrams both were well-formed and representative of the WSL code that these diagrams depict. An example, the checklist includes, among the conditions to be checked, that all activity diagrams include both a start and end state and that each action state/activity, other than start or end states, have transitions going to and from the diagram's action states/activities. These conditions in the checklist help ensure that the generated UML activity diagrams are well-formed. Other condition in the checklist ensure that control constructs in the WSL code are modelled as guards in the UML diagrams in transitions going to their corresponding control blocks and that each WSL codeline, other than control constructs and the file record field accesses of the same record which have been consolidated into a single state, are modelled as UML action states/activties. The latter conditions help ensure that the generated UML diagrams are representative of the WSL code that they model. By using this checklist and by ensuring that the generated UML diagrams meet the conditions, where applicable, outlined in the checklist for each generated diagram, the diagrams are validated to be both well-formed and representative of the restructured WSL code that they represent. A checklist also enables others to oversee the process of UML diagram validation and to ensure that this process was followed in a way that a letter from expert user validor, who attests to the final results but who does not outline the methods followed to validate the diagrams, does not provide. Furthermore, an expert (Mr. Shellenberg), with expertise in the thesis' selected system as well as in COBOL and in UML, attested, in a letter, that his examination of the generated WSL-UML notation seems to model the attributes, relationships, and structures of the original WSL code.

In any legacy system, there are many similar code segments. If a representative code segment, such as CR750V02, is validated to be correct, the algorithms that restructured and extracted these UML diagrams from the system code will also be validated. Different types of code segments, corresponding to the different types of code patterns available in the legacy system, were selected and their output in the form of extracted UML was summarised in tables. These other code segments were not shown, via extracted UML diagrams, due to their size requirements.

11.7.4 Letter from Local Supervisor

March 18, 2005

Letter from Local Supervisor/Validator

I am an experienced Information Technology professional with over 25 years of Information Technology experience, including 21 years in the telecommunications company used in this study. Consequently, I am both experienced and very familiar with this company's systems, including main-frame batch-oriented systems such as the selected legacy system used in this thesis. I also have expertise in COBOL and UML technologies that I have used in the past. With this background, I am able to validate the COBOL to WSL translation and WSL-UML notation modeling processes.

Because the research result of this thesis has a large amount of applicability to the information technology industry and because of my industry background, I am able to provide an industry perspective to the thesis' proposed approach in transforming certain legacy systems and the benefits that this approach might bring to the industry.

Although I acted as the local supervisor for **Richard Millham**, I evaluated the thesis' approach and its results in an objective and professional way.

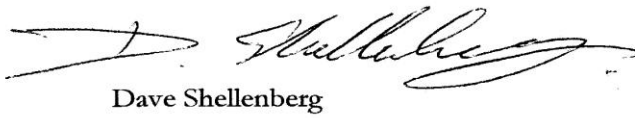
I immediately recognized, from my personal experience at the company, the thesis' selected legacy system as part of the telecommunication company's core set of systems, which handled Customer Service Orders. This system was a batch-oriented COBOL system that ran on an IBM 370 mainframe computer at one of the telecommunication company's data centres.

Given a limited selection of the COBOL samples and their corresponding translation into WSL, the COBOL to WSL conversion process appears to effectively represent a significant portion of the original source code. The WSL to UML notation conversion process appears to model the processes, relationships, and structures of the WSL representation and of the original COBOL code. An example, the WSL to UML notation conversion captures many of the classes and attributes that would be derived, through the WSL intermediate representation, from the original COBOL source code.

Viewing the thesis' approach from an industry perspective and from my experience, I feel that the thesis' proposed approach seems to be well thought out and logical. I do recognize that the COBOL to WSL conversion process will not be totally complete, nor will the WSL to UML conversion process. From my industry experience, I do know that attempting to translate a procedural based application into an object based application is no simple task. The results obtained from the thesis's tool and thesis's approach provides a 'good' starting point for a significant application reengineering effort.

Speaking from an industry standpoint, because there still exist many procedural legacy applications in various companies, a validated method of assisting in the reengineering of these applications to object-orientated implementations could be of strong financial benefit to these companies.

Sincerely,



Dave Shellenberg

Mr. Shellenberg's letter attests that from an industry standpoint, there exist many procedural legacy systems in many companies where a validated method of reengineering procedural to object-oriented systems could be of strong financial benefit to the companies that own these systems. Consequently, the thesis' restructuring process could be of large financial benefit to companies that utilise it. By providing visualised views to COBOL programs in as standardised format, this visualisation helps the developer understand the program without a steep learning curve for proprietary software reengineering/re-documentation tools. Unfortunately, Mr Shellenberg was unable to provide further input in regards to this system, particularly to the relationship

between the extracted diagrams and their direct usefulness to the system's developers. Mr Shellenberg had left the telecommunications company from which this thesis' selected legacy system was obtained and he was too busy with other work to provide any further industrial feedback. Given this situation, parallel studies were found to demonstrate the effective of program re-documentation using tools similar to TAGDUR.

In addition to Mr. Shellenberg's statement, there are several parallel studies on the benefits of re-documentation. One of these case studies involved RIGI, a reengingeering tool, that is similar to TAGDUR in that in restructures and re-documents legacy systems. RIGI re-documented, based on automatic extraction of information, a one million line database system; the extracted documentation produced a more accurate operational representation of this system than the current documentation based on a change log and developers' memories. However, RIGI, unlike TAGDUR, is semi-automatic and requires developers to write scripts to extract system software artefacts for representation within the tool. These scripts required two days of effort on part of the developers in order to develop these scripts but the scripts, once produced, enabled re-documentation of the system source code to be generated within minutes. This documentation was a remarkable savings in the system analysts' time compared to time that would be take with the manual approach of reading program listings and consulting existing program documentation (Wong, 1995). PAS, a hypertext-based software source code documentation system which documented both modules and functions of a system, was used to re-document changes to a software system, a data communications services, and the re-documentation's effects were studied over a four year period. One noticeable effect of the re-documentation, and the subsequent increased program understanding of the current code, was the reduction of the average minutes of maintenance effort per component change from 315 minutes, in 1997 at the start of the study, to 103.2 minutes in 2000, near the end of the study. This re-documentation's effect was suprising given that the developers were already very familiar with the system code (Rajlich, 2004). The purpose of these two studies is to demonstrate that re-documentation, with the subsequent increased program understanding, of legacy systems has proven to result in large cost and time savings and in an increased accuracy of program representation.

11.8 Summary

TAGDUR was designed to restructure a batch-oriented legacy system, translated into a WSL representation, into an object-oriented paradigm and then model this restructured system using a series of UML diagrams.

By selecting a several representative samples of the selected legacy system, it can be demonstrated that the thesis tool can restructure and re-document a batch-oriented system, regardless of the differing sizes, programming styles, amount of control logic, and ratio of variable declaration versus procedural code that are present in these samples.

Chapter 12: Achievements, Discussion and Conclusion

12.1 Introduction

In order to modify a legacy system to meet new business needs, developers must fully understand the systems being integrated. They must not only understand the software structure of the system but be able to follow the data and control flow and the execution of events (Ulrich, 2000). Documentation detailing this information for most legacy systems is either missing or out-of-date (Kwiatkowski, 1998). Consequently, any reengineering efforts must first have some facility to generate documentation regarding the structure and dynamics of this system.

Understanding a system is of critical importance to a developer. A developer must be able to understand the business processes being modeled by the system along with the system's functionality, structure, events, and interactions with external entities (Nelson, 1996; Wong, 1995).

This understanding is of even more importance in reverse engineering. Although developers have the advantage of having the source code available, system documentation is often missing or incomplete and the original users, whose requirements were used to design the system, are often long gone (Kwiatkowski, 1998).

This thesis is a case study of first restructuring a selected batch-oriented procedural system into an object-oriented paradigm, and secondly re-documentating this system, through a series of UML diagrams, through static analysis of its code. This re-documentation may serve as a guide for developers to redevelop the system into an object-oriented system or in the redevelopment/redeployment of this system to a new platform. A C++ generator, not fully developed, could automate this task, in the future, and provide round-trip reengineering for a procedural legacy system.

The thesis tool provides no stimulus-response trace mechanism, partially due to problems associated with such a dynamic trace such as incomplete scenarios and ambiguous traces (Egyed, 1999) and partially due to the batch-oriented nature of the selected system. Consequently, a system that critically relies on states and events to depict their functionality, such as reactive real-time systems, would not be suitable for re-documentation provided through this thesis tool. However, other types of systems, such as interactive non-reactive systems, could adopt this tool for its re-documentation purposes and they would benefit from the peripheral identification part of the generation of the system's UML component diagram. Because the selected legacy system of this thesis lacked its original user interface, the extracted UML component diagram did not depict any external peripherals, other than a file system, although the tool had the capability to do so.

The restructured system, by encompassing each class object in its own error handling mechanism, provides a guide for developers to redevelop the system in a more robust manner since the original system lacked any error-handling mechanism. The C++ generator also encompasses error handlers in its generated code and in its rules for code generation (see Appendix D).

In this chapter, an evaluation is provided of the strengths and weaknesses of the thesis tool's methods, answers to the thesis' research questions are provided, and areas for future work are identified.

12.2 Evaluation and Analysis of Thesis' UML Diagram Extraction Methods

Our modeling of the system through UML diagrams is based on analysis of system code. This code based extraction of generated UML diagrams has both advantages and disadvantages.

One advantage is that it is well suited to systems, such as ours, where the only system artefact is software code with no input available from system users. Another advantage is that new documentation can be generated for each change in code.

One disadvantage is that user input plays an important part in capturing and then representing system information. Limiting system representation to source code greatly constrains the ability to fully represent the system. An example, the TAGDUR tool can represent the business processes' functionality modeled in the system by analysing its source code, yet the tool, without system documentation or user input, can not determine the name and function of external actors of the system. Because of this inability to determine external actors, use case diagrams, which incorporate external actors as an important component of their diagram, can not adequately be derived from source code. Use case diagrams, consequently, can not be generated solely from source code.

Statecharts that model states, transitions, and events can not be adequately represented in this selected batch system. There is no clear correspondence between code and system states. States do not just represent the activities executed by code but also states represent all possible states experienced by the system as the activities of code interact with external events of the operating system and external actors. This complex interaction can not easily be broken down into a series of states by statically analysing the system's source code. The use of static information is very useful in understanding the design and structure of a system, but reveals nothing about the behaviour of the system at run-time. In order to understand this behaviour, several techniques, such as method wrapping or logging, are used to generate a trace, which contains information about the run-time behaviour of the system. Systa (Systa, 1997) had highlighted a method of using a small but representative set of data to represent these events and then record the system's response to these events. In this manner, the events and the system states could be depicted. However, given that no data is available (due to confidentiality reasons, even sample data) and the batch-oriented nature of this system, a statechart to depict system behaviour would not be possible to extract nor particularly of any use to the developers in order to better understand the system being modelled. Egyed recognises the problems of dynamic trace such as ambiguous traces and possible lack of representative scenarios (Egyed, 1999). One limitation of this trace is that it is very difficult to infer higher-level information from a trace. An example, the developer may want to know which other objects a certain object is sending information to but he does not want to analyse and verify every single method invocation to do so (Bertuli, 2003). Events, as

identified in this system, are internal events such as procedural and I/O calls along with possible system and error interrupts. Other than the arrival of input files or the output of processed files, no external events occur. If an error or system interrupt occurs, the system, with no interrupt or error handling code, simply enters into an unknown state.

Consequently, statecharts are not produced; instead, activity diagrams, which can more easily be derived from the source code of a batch-oriented system, are used instead of statecharts to represent the dynamic behaviour of a system. Procedures that involve calling other procedures have their procedure calls depicted as as sub-machines whose behavior can be broken down into the called procedure's individual codeline activities, if necessary.

Furthermore, the granularity of the activity diagrams is very fine. Each activity is modeled based on a WSL code line. Although WSL is platform-independent, and as such does not involve platform-dependent functions such as the allocation and deallocation of memory, which avoids much of the platform-dependent detail that can clutter up an activity diagram, the activity diagrams are quite detailed. Variable assignments are modeled as activities in order to avoid missing important information as to how the state space is modified. When records are accessed, either in a read or update operation, each record field access is consolidated into its parent record structure such that these record accesses are modelled as a single action, which simplifies the UML activity diagram, in light of the frequently large number of record fields within a COBOL record.

The granularity of sequence diagrams is more coarse. Each sequence is based on events, including procedural calls, among objects rather than individual codelines. TAGDUR, through its sequence diagrams, outlines the method invocations of objects in a visual way such that the developer can more easily locate the desired method invocation in a diagram.

The goal of this thesis' reengineering effort in regards to the UML diagram representation is to automatically generate UML diagrams, to the degree possible using source code as the only possible system artefact, from source code. Once this series of UML diagrams has been generated, the developer is free to modify and add information about the system, such as external actors, that can not be easily derived from its source code.

As the thesis analysis and experiments show, the type of legacy system and the use of WSL greatly constrains the type of UML diagrams that can be obtained and the methods that could be used to extract them from legacy code. Our selected legacy system was a standalone telecommunications system written in the 1960 and 1970s by a small group of users and developers that are no longer with the company. The original documentation is missing. The original system was batch-oriented, designed to operate via batch cards handled by company employers. During the 1980's, the awkward COBOL interface was replaced by Visual Basic screen scrapers that communicated with this system via a series of user request/response files placed in specified locations. Its long maintenance history obfuscated its original design and purpose. Furthermore, it is characterised by a number of small files (105 COBOL files) with many small procedures, large COBOL records, and awkward control logic. This system has no interrupt logic and very little, if any, error-handling capability.

Use case diagrams are particularly difficult to reengineer, especially based on source code analysis. Although it is possible to extract business rules and processes from the source code (Aiken, 2000), these business rules and processes need to be validated by the developer/user community. Furthermore, although business processes can be modelled, the external actors to this system

need to be identified and this identification is impossible relying just on source code. A developer/user community must be available in order to identify these external actors and the roles that they play in regards to the system or these actors with their roles must be available in up-to-date documentation; in this case, neither documentation nor the developer/user community exists. Consequently, use case diagrams, based on source code, as in this sample system, can not be obtained.

Classes may be obtained from procedurally-structured systems. However, ensuring that the correct classes are identified along with their related variables and procedures is difficult, relying on source code alone. The best bet is to choose an algorithm that clusters closely coupled variables and procedures together into classes and use these classes to restructure the original procedurally structured system. The thesis has demonstrated that this has been accomplished through an analysis of intra-class events and procedure calls. Classes form the basis of any object-oriented system and an object-oriented system is a pre-requisite for any modelling in UML.

This legacy system restricts the type of approach that can be provided in the extraction of component and deployment diagrams. Since all interactions are via files rather than a user interface (post 1980s), the original deployments, as indicated by the source code, can not be extracted. Using the thesis' component extraction method, it is possible to extract physical devices attached to software components in legacy systems with user interfaces (this has been demonstrated with a sample system file with several peripheral devices) but it is not possible to do so in this sample legacy system. There is a mapping of logical filenames within the software components of this system to physical filenames or devices. This mapping is of use to developers to ensure that although software components may read or write to different logical file names, they may really be interacting with the same physical device.

12.3 Conclusions to Research Questions

The original research questions were as follows:

3) Given a procedurally-structured, batch-oriented COBOL system, is it possible to develop algorithms, whose output is validated through a UML checklist to ensure diagram syntactical correctness and representational accuracy, to re-document this system, by extracting from the source code, a series of UML diagrams that represent the structural, behavioural, dynamic, and architectural views of the system?

4) Does this re-documentation effort entail restructuring and entity identification/determination processes? If so, is it possible to develop algorithms, validated through various means, that handle these processes and how can the information, obtained during these algorithms' execution, aid in the extraction of these UML diagrams?

5) If this system could be re-documented, is it possible to demonstrate, through parallel case studies of other re-documentation/reengineering tools, that this re-documentation of the system, by achieving a better system understanding, increases the effectiveness of the developers in maintaining this system?

In regards to question 1, it was determined that it was necessary to restructure a procedural system to an object-oriented one partially in order to re-document the restructured system, using a standardized object-oriented notation, UML, and partially for the advantages of object-orientation such as easier maintenance and easier platform redeployment. This restructuring involved first converting the original source code to a formal intermediate language representation whose advantages include its

extensibility, its mathematical foundation, and its programming language independence which made many of its restructuring algorithms, such as object clustering, not dependent on its source programming language constructs (Zhou, 2003). Once this conversion to WSL has been completed, algorithms were developed to identify potential object clusters, evaluate tasks at varying granularity levels for their execution algorithms, and identify events from the WSL code representation. These algorithms have been validated through different means: the object clustering algorithm was validated through a renaming proof and the independent task algorithm was validated against a carefully selected set of test cases. The information gained during the restructuring process was used directly in the UML extraction process. An example, the objects identified during the object clustering phase are used when modeling UML class and sequence diagrams. The tasks identified as parallel or sequential are used when modeling parallel or sequential flows in UML diagrams or to formulate sequence numbers assigned events in a UML sequence diagram. The events identified from the WSL code are represented as events in the UML sequence diagram.

In regards to question 2, algorithms were developed to extract class (structural view), activity (behavioural view), sequence (dynamic view), and component (architectural view) algorithms from the WSL code. WSL was extended to represent the abstract modelling notations of UML in WSL-UML notation. The UML diagrams that have been extracted from a selected code sample from the system, which is representative of many COBOL, batch-oriented code segments, have been validated, by a UML checklist (see Appendix H), to be representative of the code sample and syntatically correct as per the UML modelling rules.

In regards to question 3, parallel case studies were used to demonstrate that the use of similar restructuring/redocumentation tools, such as RIGI, achieved significant time savings for systems analysts in understanding their system's structure and functionality over the conventional approach of analysing system code and documentation. Furthermore, RIGI's redocumentation proved to be a more accurate depiction of the system's functionality than that derived from the maintainers' memories (Wong, 1995). Another long term case study of a redocumentation tool, PAS, demonstrated a final 324% savings in maintenance time for maintainenance done in conjunction with the tool compared to maintenance done without the tool's aid (Rajlich, 2004).

12.4 Success and Challenges

Besides the success criteria mentioned above, it is believed that this project had additional successes as well as some challenges.

By analysing and recording the data flow of the system, in the form of what variables are used as conditions within control constructs or what variables/values are assigned to other variables, it is possible to determine the following:

a) What data is used where within the system and under what conditions.

b) the recording of rules, as contained within their condition constructs and the actions that are performed when these conditions are met, as contained within the control blocks. These rules may be analysed in order to determine the set of business rules that govern the system (Aiken, 2000).

Knowledge about the structure, behaviour, interface, and other facets of a legacy system, which this thesis' tool provides, is necessary even if the goal of this reengineering effort is not to modify the system to meet new business needs or restructure this system to fit in better with a new platform. An example, if this system is simply enclosed in an object wrapper in order for it to be used within a component-based Web environment, the developers would need to have a strong knowledge of this original system, such as its structure, its behaviour, and its interface, in order to develop and then implement this object wrapper. Our tool,

through its series of UML diagrams, is able to provide much of this information and in a more comprehensive and speedier manner than if these developers had to extract this information from a manual analysis of legacy source code.

There is a nascent ability to generate C++ code from the WSL representation and from information gained during the COBOL to WSL conversion phase. This ability needs to be further developed but has several advantages. COBOL has many similar variable names, which during manual conversion, are often subject to mis-conversion errors. An automatic generation of C++ code reduces these conversion errors considerably. Another advantage is that the WSL representation is already restructured and modelled as an object-oriented system; there is no need to manually convert the COBOL code from a procedural to an object-oriented structure. The third advantage is that manual conversion of code, particularly large masses of code, involves a large expense and a large amount of time. Automating this conversion reduces this cost, both in terms of time and money considerably, although this automated conversion probably would require some manual rework.

COBOL to WSL conversion was handled via a manual process using a set of rules that were developed using the body of literature available on WSL. Efforts were made to automate this conversion, using a number of different ways such as the YACC tool or further development of existing parsing tools. This conversion process was restricted by a number of factors such as the huge number of COBOL constructs and dialects, the lack of freely-available COBOL grammars that could be used by compiler-generators and which would work on the particular COBOL dialect of this sample system. Jan Kwiatkowski developed a partial COBOL to WSL translator using YACC; however, this translator did not work with the COBOL dialect of the sample system. Incorporating a COBOL to WSL translator within the tool, even for a particular COBOL dialect, was hampered by the large number of COBOL constructs present that would have to be accommodated within the tool and the difficult method to convert COBOL file system syntax to WSL. Although there are many legacy systems written in COBOL, for the above-mentioned reasons, developing a translator from COBOL to another language is a very difficult and onerous task. Although this problem has not been thoroughly investigated, personal experience in this area would indicate that any such work would require a multi-phase transition where unstructured COBOL is structured to structured COBOL, equivalent COBOL constructs (such as GIVING and EQUALS) are converted into a common syntax, and various dialects of COBOL are converted into a common COBOL dialect even before the COBOL to another language translation begins.

One of the challenges faced was the insurmountable difficulty in obtaining feedback from developers/experts of the selected legacy system in both validating the generated UML diagrams (see section 11.7.3) and in assessing the effectiveness of this tool in an industrial setting. In response to these difficulties, alternative strategies of validation and assessing the tool's industrial effectiveness had to be used. In regards to validation, a UML checklist, which documented the process an expert would use in validating these diagrams, was applied against the thesis' generated diagrams in order to ensure that these diagrams were syntactically correct and that these diagrams were representative of their restructured WSL code (see section 11.7.3). In regards to the practical industrial effectivness of the tool, parallel case studies of a similar restructuring/re-documentation and a re-documentation tool were used to demonstrate that visual documentation of the legacy system has significant savings in industry over conventional means of analysing source code and documentation (see Section 11.7.4). Section 11.6 outlines several examples of how the visual UML diagrams produced by the thesis tool could aid the developer in migrating this selected legacy system to a Web environment. The inability to obtain evidence of the industrial effectiveness of this particular thesis tool is outlined in Section 12.5 and it is outlined also as a possible area for a future case study in Section 12.6.

12.5 Conclusions

The reengineering of a legacy system, and the development of tools to support this reengineering, is a large and difficult effort. This reengineering involves many issues related to code reengineering such as data reengineering, data typing, formal methods, et al. Bisbal recognises that a huge variety of legacy systems possible along with their potential impacts on different areas of software engineering such as program and data understanding, target system development, et al. Bisbal also recognises that a proper understanding of the legacy system is crucial because if a misunderstanding occurs, errors in this understanding are transferred to the new targeted system (Bisbal, 1999). This thesis, although it recognises the need for data understanding and reengineering, focuses on program understanding. This program understanding focuses on of one of the possible types of legacy systems, batch-oriented systems, and relies on reengineering using system code as its only system artefact.

One part of this thesis' reengineering effort was the restructuring of the legacy system from a procedurally structured and procedurally driven system to object-oriented, event-driven one. This restructuring was needed due to several factors such as the need for object orientation for redeployment to a different platform; for easier software understanding and maintenance due to object orientation; and for the identification of events and of parallel data and control flows for UML diagram extraction. This legacy system, after the conversion of the selected legacy system into the formal intermediate representation of WSL, is transformed into an object-oriented structure using a validated set of restructuring processes.

One of the aspects of this thesis is the restructuring of a legacy system into an object-oriented structure. Most reengineering/re-documentation tools focus on the re-documentation or reengineering of a system whose end result is a structure very similar to the original one. Often, this original structure is antiquated such that many modern tools that assume an object-oriented structure can not model its structure. Furthermore, object orientation has many benefits such as easier system understanding through modularization, lower maintenance costs, and easier deployment to more modern architectural platforms such as Web platforms. These benefits are not realised when the system is in its original structure.

The thesis' object orientation algorithm was a refinement of algorithms of Newcomb, Gall, and van Deursen (Newcomb, 1999; Gall, 1998; van Deursen, 1999a) as seen in Chapter 6. As demonstrated in the experiments of (Millham, 2002), if the elimination of procedures with high fan-in and high fan-out were not excluded during the object identification phase, the resulting objects that would be identified resulted in fewer objects with functional cohesion, the weakest form of cohesion, linking the procedures in the class. The results of the event identification experiments confirmed the tight cohesion and coupling of the objects identified using this thesis' object identification algorithm – most of the inter-class method invocation and attribute usage was either linked to either external devices such as files or to logging functions.

The thesis' independent task evaluation algorithm (Millham, 2003a) demonstrated that the selected legacy system, for the most part, consisted of small procedures with highly linked variables, either in control logic or in data usage. Possible parallelisation of this system, at various granular levels, is difficult. One of the areas that the independent task evaluation algorithm identifies as a possible area for large amount of parallelization is in the area of file operations. In the selected legacy system, the records are quite large. Although theoretically each file operation, without a data or control dependency, can be executed independently, the implementation of these operations, in parallel, depends on the locking level of granularity of the particular underlying database

system and on the particular architecture. This area of file operation highlights the need for data reengineering – the splitting up of these large records into smaller, logically related operations whose file operations, provided that these operations are executed on separate records, often can be executed in parallel without violating implementation-level granular (table) locking.

Another feature that this independent task evaluation algorithm discovered about this legacy system was that it was able to determine, from its data dependencies, that this system, other than its large file operations, did not possess numerous large atomic operations. Instead, it consisted of a series of sequences of tightly coupled data manipulations. This coupling consisted of data used in assignments, control constructs, and file operations. While sequential processing of data can be expected in a batch-oriented system (by definition), this coupling also is indicative of embedded business rules within the system as data is manipulated and then used by its control constructs to govern the system's execution flow (Aiken, 2000).

Business analysts can view the data manipulation and control constructs, at a low level of detail, in the extracted activity diagrams or, at a higher level of detail, in the sequence diagrams. Additionally, this data manipulation is recorded in a relational database of the TAGDUR tool such that a business analyst, through queries, could determine which variable is used in a control construct or operand, in what manner, and in what places in the system. Through such queries, it is possible to relate the data and control flows, represented by various variables, to various business rules and processes that are modelled in the system. It is important to understand the various business rules and processes that are being modelled by a system before it can be redeveloped to meet new or changing needs or even redeployed via object wrapping (Aiken, 2000). Object wrapping of a legacy system assumes that the functionality and the embedded business rules of the legacy system correctly model the current requirements of this system; often, this is not the case. Hence, changing business rules and system functionality entail the need for re-documentation of existing legacy systems in order for their redevelopment to meet new changing requirements and new or changed business rules.

The extraction of UML diagrams from a system, which forms the thesis' re-documentation of the transformed legacy system, satisfactorily depends on a number of key factors. These factors include things such as existing artefacts, the presence of a supporting user/developer community, and on the nature of the legacy system being reengineered. In this thesis, the focus was on the extraction of UML diagrams from a batch-oriented system.

Some types of UML diagrams such as use cases can not satisfactorily be extracted from source code but are highly dependent on user interaction to identify external actors and their roles. Other types, such as statecharts, are suited to model the behaviour of highly reactive systems with many identifiable external events. In the case of this thesis' selected legacy system, a batch-oriented system, an UML activity diagram is better suited to describe the system's behaviour.

Re-documentation of a legacy system is significant for many reasons. Often, reengineering is motivated by major changes in requirements of the legacy system's functionality or the need for redeployment to a different type of platform. Re-documentation provides a guide to developers to restructure the original system into an object-oriented system. Re-documentation, by depicting the data and control flows within the system, enable analysts to extract the business rules embedded within the system and review these extracted rules against new and changed requirements (Aiken, 2000).

Although data reengineering is outside the scope of this thesis, it is recognised that data reengineering is often necessary to both improve, in terms of lower maintenance costs, the legacy system and enable this legacy system to meet new platform and business requirements. Because program and data understanding are often so inter-twined, it is possible, with a good program understanding of data flow, to try and derive some of the meaning, significance, and possible range of values of various pieces of data inherent within a legacy system. However, this type of secondary, derived data understanding is outside the scope of this thesis.

Our case study confirms that it is possible to extend WSL to represent a type of object-oriented system and then model this system in a selected series of UML diagrams, in WSL notation. By closely relating this thesis' WSL-UML notation to UML Specification 1.5, there is assurance that this notation is UML-compatible and hence, this notation is easily convertible to the various UML exchange formats used by different UML modelling tools. Because UML is a well-established industry standard, well-known by many industry developers, with good tool support, the system, as represented by UML diagrams, have a better chance of being properly understood by industry developers than a formal notation. Although UML, by its nature is imprecise due to its sometimes ambiguous notation (Paige, 1999), the advantages of its wide adoption and easier understanding by industry developers outweigh the loss of precision than if its formal notation of WSL, its intermediate representation, was retained.

Besides modelling a system in an industry-wide standard, UML, rather than a lesser known formal notation which would entail a steeper learning curve for developers, the tool, TAGDUR, by automating the extraction of a model, as represented through a series of UML diagrams, from system source code overcomes two additional problems with formal modelling of a system: the typically long time that it takes to model a system using manual intervention and using a possible series of tools and the difficulty in quickly remodelling the system after system changes. TAGDUR is simply run on the changed system and a new set of UML diagrams are extracted. The old and new set of UML diagrams can be compared for any foreseen and unforeseen changes to the system. Several disadvantages of this automated approach are that it excludes any user expertise to refine and clarify these diagrams. The idea is that these users, if available, would be able to refine the diagrams after their automated extraction. This automation produces a rather simplistic extracted diagram representation of a system, such as a one object per class representation.

One of the difficulties encountered was the inability to provide practical evidence of the advantages to business in adopting the reengineering approach outlined in this thesis, particularly in regards to the thesis' selected legacy system. The reasons for this inability to obtain practical evidence is outlined in Sections 12.4 and 11.7.4. However, although such evidence is unavailable, UML diagrams were extracted from the legacy system and validated using a UML checklist (see section 11.7.3), and independent evidence was provided as to the usefulness of visual documentation (see section 11.7.4).

Our thesis is an attempt to reduce some of the problems inherent with legacy system migration through program understanding of this legacy system. Because of the huge array of possible legacy systems, this thesis focuses on batch-oriented systems with system code as its only form of documentation. These types of systems and single source of software artefact are common enough among the wide range of legacy systems. Through re-documentation, which provides program understanding, it is hoped to reduce some of the problems of program misunderstanding inherent in this migration.

12.6 Future Work

There are a number of areas of future work that can be pursued:

- The tool, TAGDUR (Transformation And Generation of Documentation through UML and Reengineering), can be upgraded to be more robust and to handle industrial-sized projects.
- A stimulus-response graphing approach could be further developed and incorporated into this tool in order to enable this tool to handle different legacy systems, such as highly-reactive legacy systems rather than batch-oriented systems
- C++ generation that this tool has the nascent ability to provide can be improved and extended in order to reduce much of the drudgery in reengineering an existing system. An accurate system model of a legacy system is crucial for developers to more fully understand its structure and behaviour in order to accomplish their tasks such as recoding it to meet new business needs or integrating it with existing systems. In this recoding, there is much drudgery, along with the possibility of difficult-to-detect errors, when you are dealing with existing COBOL names, which tend to be very long and very similar. One proposed solution of renaming these COBOL names to shorter names is fraught with the danger that much of the domain knowledge, inherent in these names (Pu, 2003), would be lost. An automated tool to translate the COBOL from WSL to C++ would greatly reduce the amount of time needed to reengineer such a system and would greatly reduce errors while still retaining the old COBOL names and their inherent domain knowledge. Underlying this translation from a COBOL to a C++ representation would be data reengineering. COBOL is very specific in terms of the exact length of its fields; C++ has a set length for its fields. Converting COBOL records to C++ records would involve much data conversion.
- Data reengineering is an important part of any reengineering effort but data reengineering is presently out of the scope of this thesis. One of the problems with COBOL legacy systems is that new fields are appended to existing records rather than split into smaller, more logically related records. As a result, COBOL records contain much irrelevant data that could be better put in a more logical relational data model. This data reengineering involves not only the reengineering of the underlying database and its records and various entities but also such entities such as triggers, COBOL code segments that access this database, and existing data conversion.
- Extraction of domain knowledge, other than UML diagrams, was outside the scope of this thesis but also a very relevant area. Pu (Pu, 2003) identifies file, procedural, and variable names as containing a type of domain knowledge. An example, the variable ACCT-BAL could indicate that the variable contains information regarding a particular person's bank balance. However, the lack of access to information, whether from documentation or from developers' insight, along with many different developers making incremental changes over many years, make the interpretation of this source code, such as variable names, difficult and prone to errors. No parsing of comments in order to provide relevant information about the program was attempted. One reason for this non-parsing of comments, and the resultant gain in domain knowledge that such a parsing would produce, is that there is no consistency in natural language used to represent a comment. Compounding this problem with natural language is that many different developers commented this selected legacy system, each with their own method of commenting the same situation in code. Furthermore, domain knowledge regarding the use of variables and procedure may be used to find more cohesive objects from the source code.
- In order to better determine the industrial effectiveness of the thesis' tool and its methods, a new case study could be undertaken using the thesis tool. This case study may involve a different batch-oriented system and another organization. However, it is crucial that once the thesis tool's generated UML diagrams are available from a selected legacy system, a study be conducted, using these diagrams with their program understanding effect, on the developers of the system in quantifying any decreased maintenance time, reduced errors, or reduced effort in redeployment to another platform.

References

(Aebi, 1997) Aebi, D "Data Reengineering: A Case Study", ed. C.J. Rijsbergen, *Proceedings in Advances in Databases and Information Systems*, Springer Verlag, Berlin, Germany.

(Aho, 1986) Aho, Alfred, Jeffrey Ullman Principles of Compiler Design Addison-Wesley, Reading, MA, USA.

(Aiken, 2000) Aiken, Peter, Chip Didden "Business Rules Extraction: A Case Study", *Third Business Rules Forum*, USA.

(Alexander, 1977) Alexander, C, S Ishikawa, M Silverstein, M Jacobson, I Fiskdahl-King, S Angel A Pattern Language, Oxford University Press, New York,USA.

(Alhir, 1998) Alhir, Si Sinan UML in a Nutshell O'Reilly, Sebastapol, CA, USA, p. 197.

(Ali, 1998) Ali, Jauhar and Jiro Tanaka "Implementing the Dynamic Behaviour Represented as the Multiple State Diagrams and Activity Diagrams", *Proceedings of the Third World Conference on Integrated Design and Process Technology 1998 (IDPT'98),* Vol.4, Berlin, Germany, pp.281-288.

(Alvin, 2004) Alvin, Chris "Chapter 2: Lecture Notes - Data Types", Available at <http://www.cs.wisc.edu/~calvin/cs110/Chapter2.html>.

(Arango, 1986) Arango, Guillermo, Ira Baxter, Peter Freeman, Chris Pidgeon, "TMM: Software maintenance by transformation", IEEE Software, 3(3), IEEE Press, Piscataway, NJ, USA, pp. 27-39.

(Ashrafuzzaman, 1995) Ashrafuzzaman, Mohammad "Scope and Tasks of Reverse Engineering" Available at <http://www.cs.usask.ca/homepages/grads/moa135/856/RE/node3.html#SECTION00030000000000000000>.

(Aiken, 1996) Aiken, Peter H. Data Reverse Engineering McGraw-Hill, New York, USA.

(Aiken, 2000) Aiken, Peter, Chip Didden "Business Rule Extraction Metrics: A Case Study*", Third Business Rules Forum*, USA.

(Alhir, 1998) Alhir, Sinan Si "What is the Unified Modelling Language" , Available at <http://home.earthlink.net/~salhir/whatistheuml.html>.

(Ammarguellat, 1992) Ammarguellat, Z "A Control-Flow Normalization Algorithm and Its Complexity", *IEEE Transactions on Software Engineering*, Vol 18, No 3, IEEE Press, Piscataway, NJ, USA pp. 237-251.

(Babcock, 2005) Babcock, Charles "Closing the Developer-User Gap", *Information Week* , Skokie, IL, USA, July 11.

(Balanyi, 2003) Balanyi, Z, R Ferenc "Mining design patterns from C++ source code", *ICSM*, pp. 305-314.

(Ball, 1996) Ball, Thomas, Stephen G Eick "Software Visualization in the Large", IEEE Computer, Vol. 29, IEEE Press, Piscataway, NJ, USA, April, pp 33-43.

(Barstow, 1985) Barstow, David "On convergence toward a database of program transformations", ACM Transactions on Programming Languages and Systems, Vol. 7, No. 1, pp.1-9.

(Baumann, 1993) Baumann, Peter "Beauty and the Beast or A Formal Semantic Description of the Control Constructs of COBOL and its Implementation", IFI Technical Report 93.39, IFIL, Zurich, Switzerland.

(Baumann, 1994) P. Baumann, J. Fassler, M. Kiser, Z. Ozturk, and L. Richter. "Semantics-based reverse engineering", Technical Report 94.08, Department of Computer Science, University of Zurich, Switzerland.

(Bennet, 1998) Bennet, K. H. "Do Program Transformations Help Reverse Engineering?", *International Conference On Software Maintenance*, Washington, DC, USA.

(Bennett, 1999) Bennett,S, S McRobb, R Farmer Object-Oriented Systems Analysis and Design using UML, McGraw-Hill, New York, USA.

(Beevis, 1994) Beevis, D., Mr. R. Bost, Dr. B. Döring, Mr. E. Nordø, Mr. F. Oberman, Med. Chef J-P. Papin, Dr. Ir. H. Schuffel, Mr. D. Streets "ANALYSIS TECHNIQUES FOR MAN-MACHINE SYSTEM DESIGN", (Report AC/243(P8)TR/7), Defense Research Group, NATO Headquarters, Brussels, Belgium.

(Bergland, 1981) Bergland, G.D. "A Guided Tour of Program Design Methodologies", IEEE Computer, IEEE Press, Piscataway, NJ, USA,, pp 13-37.

(Bergstra, 1984) Bergstra, J.A., J.W. Klop "Process algebra for synchronous communication", Information and Control, Vol 60, Academic Press Professional, Inc. San Diego, CA, USA , pp 109-137.

(Bertulli, 2003) "Run-Time Information Visualization for Understanding Object-Oriented Systems", *WOOR*.

(Biggerstaff, 1989) Biggerstaff, Ted J "Design recovery for maintenance and reuse", IEEE Computer, Vol. 22, No. 7, IEEE Press, Piscataway, NJ, USA.

(Biggerstaff, 1994) Biggerstaff, T., B Mitbander, and D Webster "Program understanding and the concept assignment problem", Communications of the ACM, Vol. 37, No. 5, Assn For Computing Machinery , New York, USA, pp. 72-83.

(Brinksma, 1983) Brinksma, Ed "A Tutorial on LOTOS" in Protocol Specification, Testing, and Verification V, ed. M Diaz, Elseveir, San Francisco, USA, pp. 171-194.

(Bisbal, 1999) Bisbal, Jesus, Deirdre Lawless, Bing Wu, Jane Grimson "Legacy Information Systems: Issues and Directions", IEEE Software, IEEE Press, Piscataway, NJ, USA.

(Bjorklund, 2001) Bjorklund, Dag "The SMDL Statechart Description Language: Design, Semantics, and Implementation" Diploma Thesis, Abo Akademi University, Finland.

(Blaha, 1999) Blaha, M.R. "The case for reverse engineering", IT Professional, Vol 1, Issue 2, IEEE Press, Piscataway, NJ, USA, pp. 35-41.

(Booch, 1985) Booch, Grady "Object-Oriented Development", IEEE Transactions on Software Engineering, Vol. 12, No. 2, IEEE Press, Piscataway, NJ, USA, pp. 211-231.

(Booch, 1991) Booch, Grady The Object Oriented Design with Applications Benjamin/Cummings Publishing Company Inc., New York, USA.

(Booch, 1998) Booch, G, Jacobson, C, and Rumbaugh, J The Unified Modelling Language – a reference manual, Addison Wesley, Toronto, Canada.
(Booch, 1999) Booch, G. Rumbaugh, J and Jacobson, I The Unfied Modelling Language User Guide Addison-Wesley-Longman Inc, Toronto, Canada.
(Bowen, 1995) Bowen, Jonathan P., Michael G. Hinchey "Ten Commandments of Formal Methods", IEEE Computer, Vol. 28, No. 4, IEEE Press, Piscataway, NJ, USA.
(Binksma, 1983) Brinksma, Ed. "A Tutorial on LOTOS" M. Diaz, (ed.) Protocol Specification, Testing, and Verification V, Elseveir, San Francisco, USA, pp. 171-194.
(Briand, 2003) Briand, L.C., Y. Labiche, Y. Miao "Towards the reverse engineering of UML sequence diagrams", *Proceedings of the 10th Working Conference on Reverse Engineering*, pp. 57-66.
(Brodie, 1995) Brodie, M and M Stonebraker Migrating Legacy Systems: Gateways, Interfaces, and the Incremental Approach, Morgan Kaufmann, San Francisco, USA.
(Brown, 1999) Brown, Gary DeWard Advanced COBOL: For Structured and Object-Oriented Programming, Wiley, Toronto, Canada.
(Brown, 1995) Brown, B and K Wallnau "A framework for evaluating software technology", IEEE Software, IEEE Press, Piscataway, NJ, USA, pp. 39-49.
(Brown, 2000) Brown, Steve Visual Basic 6 Complete Sybex, San Francisco, USA.
(Bruel, 1997) Bruel, J.M., France, B, and Larrondo-Petrir, M "CASE-based Rigorous, Object-Oriented Methods", *Proceedings of the 1st Northern Formal Methods Workshop*, Springer eWic series, Springer-Verlag, Berlin, Germany.
(Bull, 1995) Bull, Tim An Introduction to the WSL Program Transformer University of Durham, England.
(Burd, 1997) Burd, Elizabeth, Malcolm Munro and Clazien Wezeman Analyzing Large COBOL Programs: the extraction of reusable modules, *International Conference on Program Comprehension 1997*, pp. 401-410.
(Burd, 1999) Burd, Elizabeth, Malcolm Munro "Evaluating the Use of Dominance Trees for C and COBOL", *International Conference On Software Maintenance*, Oxford, England.
(Burd, 1998) Burd, Elizabeth "Release: Reconstruction of Legacy Systems for Evolutionary Change", *WCRE*.
(Chandi, 1992) Chandi, K, Carl Kesselman "The Derivation of Compositional Programs", *Joint International Conference and Symposium on Logic Programming*.
(Chen, 1986) Chen, Y.F., and C.V. Ramanoorthy "The C Information Abstractor", *Proceedings of COMSASC*, IEEE Press, Piscataway, NJ, USA, pp. 291-298.
(Chidamber, 1994) Chidamber, S and C Kemerer "A metrics suite for object-oriented design", IEEE Transactions on Software Engineering, Vol. 20, No. 6, IEEE Press, Piscataway, NJ, USA, pp. 476-493.
(Chikofsky, 1990) Chikofsky, E.H. and J.H. Cross "Reverse Engineering and Design Recovery : A taxonomy", IEEE Software Vol. 7, No. 1, IEEE Press, Piscataway, NJ, USA, pp. 13-17.
(Clark, 1997) Clark, Tony, Andy Evans "Foundations of the Unified Modeling Language", *Proceedings Of 2nd Northern Formal Methods Workshop*, Springer-Verlag, Berlin, Germany.
(Chung, 2000) Chung, Sam, Yun-Sik Lee "Reverse software engineering with UML for Web site maintenance", *Proceedings of the First International Conference on Web Information Systems Engineering*, Vol 2, pp. 157-161.
(Corbi, 1989) Corbi, T. "Program Understanding: Challenge for the 1990s", IBM Systems Journal, Vol. 28, No. 2, pp 294-306.
(Cox, 1991) Cox, Brad J, Andrew J Novobilski Object-Oriented Programming: An Evolutionary Approach, Addison-Wesley Publishing Company, New York, USA.
(Crow, 1998) Crow, Judith "Formalizing space shuttle software requirements: Four case studies", ACM Transactions on Software Engineering and Methodology, Vol. 7, No. 3.
(Cysewski, 1997) Cysewski, Grzegorz, Sagar Pidaparthi "Case Study in Migration to Object-Oriented System Structure Using Design Transformation Methods", *Proceedings of First Euromicro Conference on Software Maintenance and Reengineering*.
(Demeyer, 1999) Demeyer, Serger, Stephane Ducasse, Sander Tichelaar "Why Unified is not Universal: UML Shortcomings for Coping with Round-trip Engineering", *Second International Conference on the Unified Modelling Language*, Fort Collins, USA.
(Dietrich, 1989) Dietrich, W, L.R. Nackman and F. Gracer "Saving a Legacy System with Objects", *Proceedings of OOPSLA*, Assn for Computing Machinery, New York, USA, pp. 77-83.
(Di Lucca, 2003) Di Lucca, G.A., Fasolino, A.R., Tramontana, P, De Carlini "Abstracting Business Level UML diagrams from Web applications", *Proceedings of the Fifth IEEE International Workshop on Web Site Evolution*.
(Duke, 1991) Duke, D "Object-Oriented Formal Specification", PhD thesis, University of Queensland, Brisbane, Australia.
(Egyed, 1999) Egyed, A, Kruchten, P.B. "Rose/Architect : a tool to visualize architecture", *Proceedings of the 32nd Annual Hawaii International Conference on System Sciences*.
(Egyed, 2003) Egyed, Alexander "A Scenario-Driven Approach to Trace Dependency Analysis", *IEEE Transactions On Software Engineering*, vol. 29, No. 2, pp. 116-132.
(Eisenbarth, 2001) Eisenbarth, T, R. Koschke and D Simon "Aiding Program Comprehension by static and dynamic feature analysis", *Proceedings of the International Conference on Software Maintenance*.
(Eisenbarth, 2003) Eisenbarth, T, Rainer Kosche, Daniel Simon "Locating Features in Source Code", IEEE Trans on Software Engineering., Vol. 29, No. 4, pp. 210-224.
(Embarcadero, 2004) Embarcadero "Describe C++ Tutorial" and "Chapter 8: The Sequence Diagram". Available at at <http://www.embarcadero.com/products/describe/Describe_documentation_pdf/C++Tutorial/Chapter_8-_The_Sequence_Diagram.pdf>.
(Etzkorn, 1997) Etzkorn, Letha, Carl Davis "Automatically Identifying Reusable OO Legacy Code", Computer, Oct, IEEE Press, Piscataway, NJ, USA.

(Evans,1998) Evans, Andy "Reasoning with UML Class Diagrams", *Second IEEE Workshop on Industrial Strength Formal Specification Techniques, WIFT'98*, Boca Raton,Florida, USA.
(Feingold, 1981) Feingold, Carl Fundamentals of Structured COBOL Programming Wm C Brown and Company, Dubuque, Iowa, USA.
(Fergen, 1994) Fergen, Hartmut et al "Bringing Objects into COBOL: Moore – a tool for migration from COBOL to object-oriented COBOL", *Proceedings of Conference of Technology of Object-Oriented Languages and Systems*, pp. 435-556.
(Flemming, 1992) Flemming, Neilson, and Hanne R. Nielseon Semantics with Applications, Wiley, New York, USA.
(Flint, 1997) Flint, E.S. "The COBOL jigsaw puzzle: Fitting object-oriented and legacy applications together", IBM Systems Journal, Vol. 36, No. 1, 1997.
(France, 1998) France, R, Evans, A, Lano, K and Rumpe B "The UML as a Formal Modelling Notation", Computer Standards and Interfaces, No. 19, pp. 325-334.
(Gall, 1995) Gall, H, R Klosch and R Mittermeir "Architectural Transformation of Legacy Systems", *17th International Conference on Software Engineering (ICSE-17), Workshop on Program Transformation for Software Evolution*, Seattle, USA.
(Gall, 1998) Gall, H.W. Eixelsberger, M. Kalan, M. Ogris, H. Beckman, B. Bellay "Recovery of architectural structure: a case study. " in Development and Evolution of Software Architectures for Product Families , ed. F. van der Linden , *Second International ESPRIT ARES Workshop*, Las Palmas de Gran Canaria, Spain, Springer-Verlag, LNCS 1429, Berlin, Germany, pp 89-96.
(Gamma, 1995) Gamma, E, R Helm, R Johnson, J Vlissides Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Toronto, Canada.
(Ganti, 1995) Ganti, N., W Brayman Transition of Legacy Systems to a Distributed Architecture, John Wiley and Sons, New York, USA.
(Gerlich, 2000) Gerlich, Rainer "An Implementation and Verification Technique for Distributed Systems", Lecture Notes in Computer Science 2, Springer-Verlag, Berlin, Germany.
(Gold, 2004) "Gold Parsing System", Available at <http://www.devincook.com/goldparser/>.
(Goldstein, 1947) Goldstein, H, J von Neumann, A Burks, "Report on the Mathematical and Logical Aspects of an Electronic Computing Instrument", Princeton Institute of Advanced Study, Princeton, USA.
(Greefhorst, 1998) Greefhorst, Danny, et al "Evaluating OO-CASE tools: OO research meets practice", *ECOOP Workshop*, pp. 486-488.
(Gunderloy, 2003) Gunderloy, Mike Developing and Implementing Web Application with Visual Basic .Net and Visual Studio .NET Que Publishing, Indianopolis, USA.
(Harel, 1988) Harel, D., Iachover, H., Naamad, A., Pnueli, A., Sherman, R. & Shtul-Tauringet, A. "STATEMATE: A working environment for the development of complex reactive systems", *Proceedings of the Tenth IEEE International Conference on Software Engineering.* IEEE Press Piscataway, NJ, USA.
(Harris, 1995) Harris, David, Howard B Ruebenstein, Alexander S Yeh "Reverse Engineering to the Architectural Level", *19th Conference on Software Engineering,* Seattle, WA, USA.
(Hausler, 1990) Hausler, Phillip "Using Function Abstraction to Understand Program Behaviour", IEEE Software, IEEE Press, Piscataway, NJ, USA, pp. 55-63.
(Hay, 1998) Hay, David "UML Misses the Boat", *OOPSLA'98 Workshop on Formalizing UML*, Vancouver, Canada.
(Hoare, 1985) Hoare, Charles Anthony Richard Communicating Sequential Process, Prentice-Hall, London.
(Hochstettler, 1999) Hochstettler, Thomas J, Barry P McFarland, Andrea Martin, Joseph A Watters Jr. "Simultaneous Process Reengineering and System Replacement at Rice University", CAUSE/EFFECT Journal, Vol. 22, No. 3, Boulder, CO, USA.
(Holt, 2000) Holt, Richard, Hoda Fahmy "Software Architecture Transformations", *International Conference on Software Maintenance*, San Jose, CA.
(Howe, 2002) Howe, D. (2002). "FOLDOC: Free On-Line Dictionary of Computing", Available at <http://foldoc.doc.ic.ac.uk/foldoc/index.html>.
(Huang, 2003) Huang, Shihong, S Tilley "Workshop on graphical documentation for programmers: assessing the efficacy of UML diagrams for program understanding", *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, pp. 281-282.
(Huggins, 2004) "COBOL Front Door". Available at <http://www.jamesshuggins.com/h/tek1/cobol.htm>.
(Ideogramic, 2004) Ideogramic "Agile UML" Available at <http://www.ideogramic.com/products/agile-uml.html>.
(Jacobson, 1991a) Jacobson, Ivar The Object Advantage: Business Process Reengineering with Object Technology Addison-Wesley, Reading, MA, USA.
(Jacobson, 1991b) Jacobson, I , F. Lindstrom "Re-engineering of old systems to an object-oriented architecture, *Proceedings of OOPSLA.*
(Jarzabek, 1995) Jarzabek, Stan "PQTL: A Language for Specifying Program Transformations", *ICSE-17 Workshop on Program Transformations.*
(Jetley, 2001) Jetley, Raoul "Formal Specification Methods: State of the Art and Future Directions", Presentation at North Carolina University, USA, Nov.
(Johnson, 1988) Johnson, R.E. and B. Foote "Designing reusable classes", Journal of Object-Oriented Programming, Vol.1, No. 2, pp. 22-35.
(Johnson, 1996) Johnson, Randolph, Kilov, H "Can a flat notation be used to specify an OO systems: using Z to describe RM-ODP constructs", *Proceedings of the 1st IFIP Workshop on Formal Methods for Open Object-Based Distributed Systems*, Chapman and Hall, pp. 407-418.

(Joiner, 1998) Joiner, J.K., W. Tsai "Reengineering Legacy COBOL Programs", Communications of the ACM, ACM Press, New York, USA.
(Jones, 1993) Jones, Neil, D, Careston K Gomard, Peter Sestoft Partial Evaulation and Automatic Program Generation, Prentice Hall, Upper Saddle, New Jersey, USA.
(Kaufman, 1990) Kaufman, L and Rousseeuw, P.J. Finding Groups in Data : An Introduction to Cluster Analysis John Wiley, Mississauga, Canada.
(Kemmerer, 1990) Kemmerer, R. A. "Integrating Formal Methods into the Development Process", IEEE Software, vol 7, issue 9, IEEE Press, Piscataway, NJ, USA, pp. 37-50.
(Koester, 2005) Koester, Mathias, Software Architect. Interview, May 5th, 2005, Gentleware Inc.,Hamburg, Germany.
(Klosch, 1996) Klosch, Rene "Reverse Engineering : What and How to Reverse Engineer Software", *Proceedings of the California Software Symposium*, University of Southern California, pp. 92-99.
(Kollman, 2001) Kollman, R, Gogolla, M "Capturing dynamic program behaviour with UML collaboration diagrams", *Fifth European Conference on Software Maintenance and Reengineering.*
(Kollman, 2002) Kollman, R., P Selonen, E Stroulia, T Systa, A Zundorf "A Study of the current state of the art in tool-supported UML-based static reverse engineering", *Proceedings of the Ninth Working Conference on Reverse Engineering.*
(Kurfess, 1997) Kurfess, Franz J, Mrinalini Lankala, Ashok Vantipalli, Lonnie R Welch "A Toolset for the reengineering of complex computer systems", *ECBS*, pp. 97-104.
(Kwiatkowski, 1998) Kwiatkowski, Jan, Ireneusz Puchalski "Pre-processing COBOL programs for Reverse Engineering in A Software Maintenance Tool", *COTSR.*
(Lam, 1992) Lam, M, S Robert, P Wilson "Limits of Control Flow on Parallelism", *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ACM, Gold Coast, Australia, pp. 46-57.
(Lammel, 2001) "VS COBOL II grammar". Software Practical Experience 31(15): 1395-1438.
(Lano, 1994) Lano, K, H Haugton "The Z++ Manual", Technical Report, Imperial College, UK.
(Lemmon, 1988) Lemmon, E.J. Beginning Logic Hackett Publishing Co, Indianapolis, USA.
(Levey, 1996) Levey, R Reengineering COBOL with Objects: Step by Step to Sustainable Legacy Systems, McGraw Hill, New York, USA.
(Li, 2001) Li, Y., & Yang, H. (2001). "Simplicity: A Key Engineering Concept for Program Understanding", *International Workshop on Program Comprehension.*
(Linger, 1979) Linger, R.C., H.D. Mills, B.I Witt Structured Programming: Theory and Practice, Addison-Wesley, Toronto, Canada.
(Lippman, 1998) Lippman, Stanley B C++ Primer Addison-Wesley, Toronto, Canada.
(Liu, 1990) Liu, S.S, N. Wilde "Identifying Objects in a Conventional Procedural Language: An Example of Data Design Recovery", *IEEE Proceedings of the Conference on Software Maintenance*, San Diego, USA.
(Liu, 1999) Liu, Xiaodong "Abstraction: A Notation for Reverse Engineering", Ph.D. Thesis, De Montfort University, England.
(Lubara, 1991) Lubara, Mitchell D "Domain Analysis and domain engineering in IDeA", in Domain Analysis and Software Systems Modelling, Ruben Prieto-Diaz and Guillermo Arango eds., IEEE Press, Piscataway, NJ, USA, pp. 163-178.
(Maguire, 1989) Maguire, M., & Sweeney, M. "System monitoring: garbage generator or basis for comprehensive evaluation system?", A. Sutcliffe & L. Macaulay (Eds.), People and Computers V. *Proceedings of the Fifth Conference of the British Computer Society, Human-Computer Interaction Specialist Group.,* Cambridge University Press, Cambridge, UK.
(Mammel, 1996) Mammel, P.I. ANSI COBOL III in SDF + an ASF definition of a Y2K tool. Master's thesis. University of Amsterdam. Programming Research Group.
(Mandrioli, 1985) Mandrioli, D., R Zicari, C. Ghezzi, F. Timsato "Modelling the Ada task system by Petri nets", Computer Languages, 10(1), pp. 43-61.
(Mansurov, 2003) Mansurov, Nikolai, Djenana Campara "Extracting High-Level Architecture from Existing Code with Summary Models", Applied Informatics, pp. 905-912.
(Meyer, 1987) Meyer, B. "Reusability: the Case for Object-Oriented Design", IEEE Software, vol. 4, no. 2, IEEE Press, Piscataway, NJ, USA.
(Milner, 1980) Milner, Robin "A Calculus of Communicating Systems", Lecture Notes in Computer Science, vol 92, Springer-Verlag, Berlin, Germany.
(Milner, 1989) Milner, Robin Communication and Concurrency. Prentice-Hall, London, UK.
(McClure, 1978) McClure, Carma L. Reducing COBOL Complexity Through Structured Programming ,Van Nostrand Reinhold Company, New York, USA.
(Millham, 2002) Millham, Richard "An Investigation: Reengineering Sequential Procedure-Driven Software into Object-Oriented Event-Driven Software through UML Diagrams", *Proceedings of the International Computer Software and Applications Conference*, Oxford, UK.
(Millham, 2003a) Millham, Richard "Determining Granularity of Independent Tasks for Reengineering a Legacy System into an OO System" *Proceedings of the International Computer Software and Applications Conference*, Dallas, Texas, UK.
(Millham, 2003b) Millham, Richard, Hongji Yang "TAGDUR: A Tool for Producing UML Diagrams Through Reengineering of Legacy Systems", *Proceedings of the IASTED Conference on Software Engineering*, Marina del Ray, USA.
(Millham, 2004) Millham, Richard, JianJun Pu, Hongji Yang "TAGDUR: A Tool for Producing UML Sequence, Deployment, and Component Diagrams Through Reengineering of Legacy Systems", *Proceedings of the IASTED Conference on Software Engineering*, Innsbruck, Austria.
(Millicev, 2002) Millicev, Dragan "Domain Mapping Using Extended UML Object Diagrams", IEEE Software, IEEE Press, Piscataway, NJ, USA, pp. 90-97.

(Morgan, 1987) Morgan, Timothy and Rami Razouk "Interactive state-space analysis of concurrent systems", IEEE Trans on Software Engineering, SE-13(10), IEEE Press, Piscataway, NJ, USA, pp. 1080-1091.
(Mosses, 1991) Mosses, Peter (ed.) "Denotational Semantics" Handbook of Theoretical Computer Science (vol. B): Formal Models and Semantics, MIT Press, Cambridge, MA, USA, pp. 575-631.
(Musuvathi, 2002) Musuvathi, Madanlal, David Y.W. Park, Andy Chou, Dawson R. Engler and David L. Dill "A Pragmatic Approach to Model Checking Real Code", *Proceedings of the 5th USENIX Symposium on Operatings Systems Design and Implementation*, Boston, USA.
(Muller, 2000a) Muller, Pierre-Alain Instant UML Wrox, Birmingham, UK.
(Muller, 2000b) Hausi A. Mller, Jens H. Jahnke, Dennis B. Smith, et al., "Reverse Engineering: A Roadmap", A. Finkelstein, (ed.), The Future of Software Engineering, ACM Press, New York, USA.
(Murphy, 1996) Murphy, Gail, David Notkin "Lightweight Source Model Extraction", ACM on Software Engineering and Methods, Vol. 5, No. 3, pp. 262-292.
(Myers, 1975) Myers, G.J. Reliable Software Through Composite Design Petrocelli Charter, New York, USA.
(Mylopoulos, 1994) Mylopoulous, John et al "Towards an Integrated Toolset for Program Understanding", *CASCON*, Toronto, Canada.
(Neighbors, 1984) Neighbors, James "The Draco Approach to constructing software from reusable components", IEEE Transactions on Software Engineering, Vol. 10, No. 5, IEEE Press, Piscataway, NJ, USA, pp. 564-571.
(Nelson, 1996) Nelson, Michael "A Survey of Reverse Engineering and Program Comprehension", ODU CS 551.
(Newcomb, 1995) Newcomb, P and Kotik, G. "Reengineering procedural into object-oriented systems", *Second Working Conference on Reverse Engineering, WCRE95 (1995)*, pp. 237-249.
(Newcomb, 1999) Newcomb, Philip "Reengineering Procedural into Object-Oriented Systems", *WCRE*, IEEE Society, Piscataway, NJ, USA.
(Newcomb, 2001) Newcomb, Philip, Doblar, R. "Automated Transformation of Legacy Systems", Journal of Defense Software Engineering, Dec issue.
(Nickel, 2000) Nickel A, Jörg Niere, Jorg P. Wadsack, Albert Zündorf "Roundtrip Engineering with FUJABA" (Extended Abstract), *Proceedings of 2nd Workshop on Software-Reengineering (WSR)*, Bad Honnef, Germany.
(Oestereich, 1999) Oestereich, Bernd Developing Software with UML: Object-oriented analysis and design in practice, Addison-Wesley, New York.
(Owen, 2004) Owen, James "Putting Rules Engines to Work", InfoWorld, Issue 25, June 28, San Francisco, California, pp. 36-40.
(Paige, 1999) Paige, Richard F "Integrating a program design calculus and a subset of UML" Computer Journal, Volume 42, Issue 02, pp. 82-99.
(Parnas, 1994) Parnas, D.L. "Software Aging" , *Proceedings of of the 16th International Conference on Software Engineering,* Sorrento, Italy, IEEE Computer Society Press, Piscataway, NJ, USA, pp. 279-287.
(Paul, 1994) Paul, S, A Prakash "On formal query languages for source code search", IEEE Transactions on Software Engineering, Vol. 20, No. 6, IEEE Press, Piscataway, NJ, USA, pp. 463-475.
(Pernul, 1995) Pernul, Gunther, Hubert Hasenauer "Combining reverse with forward database engineering – a step forward to solve the legacy system dilemma", *Proceedings of the 6th DEXXA Conference*, London, UK.
(Perumalla, 1996) Perumalla, Kaylan S, Richard M Fujimoto, Andrew T Ogieski "MetaTED – a meta language of modelling telecommunication networks", Technical Report, Georgia Institute of Technology, Atlanta, USA.
(Pidaparthi, 1998) Pidaparthi, Sagar "A Template for Design Transform Specification", *Proceedings of the International Conference of Software Engineering*, Kyoto, Japan.
(Poseidon, 2004) "Poseidon for UML" Available at <http://www.gentleware.com>.
(Price, 1993) Price, B.A, R.M. Baecker, I.S. Small "A principled taxonomy of software visualization", Journal of Visual Languages and Computing, 4(3), pp. 211-266.
(Pu, 2003) Pu, JianJun, Richard Millham, Hongji Yang "Acquiring Domain Knowledge in Systematising the Process of Reverse Engineering Legacy Code into UML", *Proceedings of the IASTED Conference on Software Engineering*, Marina del Ray, USA.
(Pu, 2005) Pu, JianJun, Hongji Yang, Steve McRobb, Richard Millham "Visualizing COBOL Legacy Systems with UML: An Experimental Report", Advances in UML-Based Software Engineering, IDEA Group, Hershey, PA, USA.
(Rajlich, 1992) Rajlich, Vaclav "Workshop Notes – Program Comprehension", Orlando, Florida, IEEE Computer Society, Miami, Florida, USA.
(Rajlich, 2004) Rajlich, Vaclav, Alexander Rostkowycz, Andrian Marcus "A Case Study on the Long-Term Effects of Software Redocumentation", *ICSM*, pp. 92-101.
(Rajput, 2000) Rajput, Wasim E-Commerce Systems Architecture and Applications Artech House, Norwood, USA.
(Rational, 2002) "Rational Rose". Available at <http://www.rational.com>.
(Rayson, 1999) Rayson, P, Garside, R, Sawyer, P. "Language engineering for the recovery of requirements from legacy documents", REVERE Project Report, Lancaster University, Lancaster, England.
(Renaissance, 1999) "Renaissance Project – Methods and Tools for the Evolution and Reeengineering of Legacy Systems" Esprit Project, Lancaster University, Lancaster, UK, 1999. Available at <http://www.comp.lancs.ac.uk/computing/research/cseg/projects/renaissance/RenasissanceWeb.htm>.
(Rich, 1990) Rich, C, and L.M. Wills 'Recognizing a Program's Design: A Graph-Parsing Approach", IEEE Software, Vol. 7, Issue 1, IEEE Press, Piscataway, NJ, USA, pp. 82-89.
(Richner, 1999) Richner, Tamar, Stephane Ducasse "Recovering High Level Views of Object-Oriented Applications from Static and Dynamic Information", *International Conference on Software Maintenance*, Oxford, England, pp. 13-22.

(Robbins, 1998) Robbins, Jason, Nenad Medvidoviv, David F. Redmiles, David S. Rosenblum "Integrating Architecture Description Languages with a Standard Design Method", *Proceedings of the International Conference on Software Engineering*, Kyoto, Japan.

(Rugaber, 1990) Rugaber, Spencer, Stephen B Ornburn, Richard J LeBlanc "Recognizing Design Decisions in Programs" in IEEE Software, Volume 7, Issue 1, IEEE Press, Piscataway, NJ, USA.

(Rugaber, 1993) Rugaber, Spencer and Richard Clayton "The Representation Problem in Reverse Engineering", *Proceedings of the 1993 Working Conference on Reverse Engineering*, IEEE Computer Society Press, Miami, Florida, USA.

(Rumbaugh, 1991) Rumbaugh, James, Michael Blaha, William Premerlani, Frederick Eddy, William Lorenson Object-Oriented Modelling and Design, Prentice-Hall, Upper Saddle, NJ, USA.

(Sannella, 1993) Sannella, Donald "A Survey of Formal Development Methods" in Software Engineering: A European Perspective, Richard Thayer and Andrew McGettrick ed., IEEE Computer Society Press, Piscataway, NJ, USA, pp. 281-297.

(Santini, 2004) Santini, Alberto "CobCy". Available at <http://web.tiscali.it/albertosantini/cobcy/>.

(Senonen, 2001) Senonen, P, K Koskimies, M Saskkinen "How to make apples from oranges in UML", *Proceedings of the 34th Annual Hawaii International Conference on System Sciences*, USA.

(Semmel, 2001) Semmel, G., G.H. Walton "Developing and validating thousands of executable finite state machines", AerospaceConference, *IEEE Proceedings*, vol 6, pp. 2837-3848.

(Siber, 2004) "COBOLTransformer Toolkit" Available at <http://www.siber.com/>.

(Siff, 1996) Siff, Michael, Thomas Reps "Program Generalization for Software Reuse: From C to C++", ACM SIGSOFT on Software Engineering, San Francisco, USA.

(Silvotti, 1994) Sivilotti, Paolo "A Verified Integration of Parallel Programming Paradigms in CC++", *8th International Parallel Processing Symposium*, IEEE, Cancun, Mexico, pp 44-50.

(Smith, 1997) Smith, Graeme, John Derrick "Refinement and verification of concurrent systems specified in Object-Z and CSP", *ICFEM*.

(Sneed, 1992) Sneed, H "Migration of Procedurally Oriented COBOL Programs in An Object-Oriented Architecture", *International Conference on Software Maintenance*, IEEE Press, Orlando, USA.

(Sneed, 1996) Sneed, H.M. "Encapsulating Legacy Software in Client-Server Systems", *Proceedings of the Third working Conference on Reverse Engineering*, IEEE Computer Society Press, Piscataway, NJ, USA, pp. 104-119.

(Sneed, 1988) Sneed, H.M. "Downsizing – Wegbereiter", Online 8.

(Sneed, 2002) Sneed, H.M. "Restructuring of COBOL/CIS Legacy Systems" , Science of Computer Programming Vol 45, No. 2.

(Srinivas, 1991) Srinivas, Yellamraju "Pattern-Marching: A Sheaf-Theoretic Approach", PhD thesis, Dept. of Information and Computer Science, University of California at Irvine, Irvine, USA.

(Stevens. 1998) Stevens, P and R Pooley "System reengineering patterns", *ACM SIGSOFT Foundations of Software Engineering (FSE-98)*, Lake Buena Vista, Florida, USA, pp. 17-23.

(Storey, 1995) Storey, Margaret-Anne D., Hausi A. Müller, Kenny Wong "Manipulating And Documenting Software Structures **",** *Proceedings of the 1995 International Conference on Software Maintenance,* Nice, France, October, pp. 16-20.

(Systa, 1997) Systa, Tarja and Kai Koskimies "Extracting state diagrams from legacy system", *ECOOP Workshops*, pp. 272-273.

(Systa, 1999) Systa, T "The relationships between static and dynamic models in reverse engineering java software", *Proceedings of the Sixth Working Conference on Reverse Engineering (WCRE-99)*, Atlanta, Georgia, USA, pp 304-313.

(Suzuki, 1999) Suzuki, Junichi and Yoshikazu Yamamoto "Making UML models interoperable with UXF", Lecture Notes in Computer Science No. 1618, Springer-Verlag, Berlin, Germany.

(Tannenbaum,1992) Tannenbaum, Andrew Modern Operating Systems Prentice-Hall, Englewood Cliffs, NJ, USA.

(Telecommunications Employee A, 2002) Oral Interviews, 1998-2002 (Name withheld for confidentiality reasons).

(Telecommunications Employee B, 1999) Oral Interview, 1999 (Name withheld for confidentiality reasons).

(Telecommunications Employee C, 1999) Oral Interview, 1999 (Name withheld for confidentiality reasons).

(Telecommunications Employee D, 1999) Oral Interview, 1999 (Name withheld for confidentiality reasons).

(Tilley, 1996) Tilley, S.R. and D.B. Smith "Perspectives on Legacy System Reeengineering", Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, USA.

(Tonella, 2001) Tonnella, P, Potrich, A "Reverse engineering of the UML class diagram from C++ code in presence of weakly-typed containers", *Proceedings of the IEEE International Conference on Software Maintenance*, pp. 376-385.

(Tonnella, 2003) Tonnella, P, Potrich, A "Reverse engineering of the interaction diagrams from C++ code", *ICSM*, pp. 159-168.

(Trauter, 1997) Trauter, Roland "Reengineering with UML". Position Paper for *ECOOP'97*.

(Ulrich, 2000) Ulrich, William "E-Integration needs Business Integration" , *ComputerWorld*, Nov 20, 2000.

(Umar, 1997) Umar, A Application (Re)Engineering: Building Web-based Applications and Dealing with Legacies Prentice-Hall, New York.

(Univan, 2000) Univan Ahn, Chris George "C++ Translator for the RAISE Specification Language", Technical Report 220, UNU/IIST, P.O. Box 3058, Macau.

(Van,1992) Van, S.L. (1992). "Workshop Notes – AI and Program Understanding", *AAAI*.

(van Deursen, 1999a) van Deursen, Arie, Tobias Kuipers "Identifying Objects using Cluster and Concepts Analysis", *Proceedings 21st International Conference on Software Engineering.*

(van Deursen, 1999b) van Deursen, Aries, Paul Klint, Chris Verhoef "Research Issues in the Renovation of Legacy Systems" in Fundamental Approaches to Software Engineering: Lecture Notes in Computer Science, Springer-Verlag, New York, USA.

(van Deursen, 2001) van Deursen, Aries "Understanding Legacy Architectures*", Landelijk Architectuur Congress*, Zeist, Netherlands.
(van den Brand, 1997a) van den Brand , Mark Paul Klint Chris Verhoef "Re-engineering needs Generic Programming Language Technology", SIGPLAN Notices of the ACM, Assn for Computing Machinery, New York, USA.
(van den Brand, 1997b) van den Brand, M.G.J., M.P.A. Sellink, C Verhoef "Obtaining a COBOL grammar from legacy code for reengineering purposes" in M, Sellink (ed), *Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications*, Electronic Workships in Computing, Springer Verlag, New York, USA.
(van den Brand, 1997c) van den Brand, Mark, Alex Sellink, Chris Verhoef "Reengineering COBOL software implies specification of the underlying dialects", Technical Report P9702, University of Amsterdam, Netherlands.
(van den Brand, 1998) van den Brand, Mark, Alex Sellink, Chris Verhoef "Current Parsing Techniques in Software Renovation Considered Harmful", *Proceedings of International Workshop on Program Comprehension*, Ischia, Italy.
(Van Sickle, 1992) Van Sickle, Larry, ed "Workshop Notes – AI and Program Understanding" , *AAAI* , San Jose, California, USA.
(Wang, 1993) Wang, Jingwen, Songnian Zhou et al. "LSBatch: A Distributed Load-Sharing Batch System", Technical Report CSRI-286, University of Toronto, Toronto, Canada.
(Ward, 1986) Ward, Paul T "The Transformation Schema: An Extension of the Data Flow Diagram to Represent Control and Timing", IEEE Transaction on Software Engineering, Vol SE-12, No. 2, IEEE Press, Piscataway, NJ, USA, pp. 198-221.
(Ward, 1989) Ward, M, F.W. Calliss, M. Munro "The maintainer's assistant", *Proceedings of Conference on Software Maintenance*, IEEE Press, Piscataway, NJ, USA, pp. 307-315.
(Ward, 1992) Ward, M., "The Syntax and Semantics of the Wide Spectrum Language", *Technical Report*, Durham University, England.
(Ward, 1994) Ward, Martin "Specifications from Source Code -- Alchemists' Dream or Practical Reality?", *4th Reengineering Forum*, Victoria, Canada.
(Ward, 2001) Ward, M. "The FermaT Assembler Re-Engineering Workbench", *International Conference on Software Maintenance*, Florence, Italy.
(Webster, 1987) Webster, Dallas E "Mapping the design representation terrain : A survey", Technical Report: MCC STP-093-87, Micro-Electronics and Computer Technology Corporation.
(Weiser, 1984) Weiser, M "Program Slicing" IEEE Transactions on Software Engineering, Vol. 10, No. 4, IEEE Press, Piscataway, NJ, USA, pp. 352-357.
(Wiggerts, 1997) Wiggerts, T., H Bosma, E Fielt "Scenarios for the identification of objects in legacy systems", *4th Working Conference on Reverse Engineering* (1997), IEEE Press, Piscataway, NJ, USA, pp. 24-32.
(Wijegunarante, 1998) Wijegunarante, Inji, George Fernandez Distributed Applications Engineering: Building New Applications And Managing Legacy Applications with Distributed Technology, Springer Verlag, New York, USA.
(Wile, 1987) Wile, David S "Local formalisms: Widening the spectrum of wide-spectrum languages" in Program Specification and Transformation, L.G.L.T. Meertens ed., Elsevier, North Holland, USA, pp. 165-195.
(Wing, 1996) Wing, Jeannette, Edmund M. Clarke "Formal Methods: State of the Art and Future Directions", ACM Computing Surveys, Vol. 28, No. 4, Assn for Computing Machinery, New York, USA, pp 626-643.
(Wirstad, 1989) Wirstad, J "The state diagram – a technique for better communication between operators and designers", *Meeting of Human Factors Society – European Chapter Meeting.*
(Wong, 1995) Wong, W. S Tilley, H Muller "Structural Redocumentation" IEEE Software, Vol 12, No.1, IEEE Press, Piscataway, NJ, USA, pp. 46-54.
(Yang, 1994) Yang, Hongji, Acquiring Data Designs from Existing Data-Intensive Programs, PhD Thesis, Durham University, England.
(Yang, 1997a) Yang, Hongji "A Design Framework for System Re-engineering", *Proceedings of Joint Asia Pacific Software Engineering Conference and International Computer Science Conference.*
(Yang, 1997b) Yang, Hongji "Formal Methods For The Re-Engineering of Computing Systems", *Proceedings of COMPSAC*, 1997.
(Yang, 1998) Yang, Hongji, Xiaodong Liu, Hussein Zedan "Tackling The Abstraction Problem for Reverse Engineering in a System Re-engineering Approach", *Proceedings of the IEEE Conference on Software Maintenance.*
(Yang, 1999a) Yang, Hongji, Z. Chen, H. Zedan, A. Cau "A Wide-Spectrum Language for Object-Based Development of Real-time Systems", International Journal of Information Sciences, Vol 118, Elsevier Sciences, London, UK, pp. 15-35.
(Yang, 1999b) Yang, Hongji et al "Acquisition of Entity Relatioship Models for Maintenance – Dealing with Data Intensive Component Indentification", *IEEE IWPC 97*, pp 148-157.
(Yang, 2003) Yang, Hongji, Martin Ward Successful Evolution of Software Systems, Artech House, Norwood, MA, USA.
(Yeh, 1991) Jen Yeh Wei, Micheal Young "Compositional Reachability Analysis Using Process Algebra", *Proceedings of the ACM SIGSOFT Symposium on Testing, Analysis, And Verification*, Victoria, Canada.
(Younger, 1993) Younger, E.J., Martin Ward "Understanding Concurrent Program using Program Transformations", *Proceedings of the IEEE Second Workshop on Program Comprehension*, Capri, Italy, pp. 160-168.
(Yourdon, 1975) Yourdon, E and L Constantine Structural Design Yourdon Press, New York, USA.
(Zahavari, 2000) Zahavi, R Enterprise Application Integration with CORBA Component and Web-Based Solutions John Wiley and Sons, Inc., New York, USA.
(Zhou, 2002) Zhou, Ying, Kostas Kontogiannis "Migration to Object Oriented Platforms: A State Transformation Approach", *ICSM*, Montreal, Canada.
(Zhou, 2003) Zhou, Ying, Kostas Kontogiannis "Incremental Transformation of Procedural Systems to Object Oriented Platforms", *Proceedings of COMPSAC*, Dallas, USA, 2003.
(Zhou, 2004) Zhou, Ying et al "Model Driven Business Process Recovery", *WCRE*, Delft, Netherlands

Appendix A: COBOL Source Sample

A sample of COBOL code from the legacy system is given, notably one medium sized.

A. CR750V02.cbl

```
IDENTIFICATION DIVISION.
 PROGRAM-ID.    CR750.
 AUTHOR.        LAWRENCE LI.
 INSTALLATION.  ALBERTA GOVERNMENT TELEPHONE.
 DATE-WRITTEN.  NOV 20/86.
 DATE-COMPILED.
 REMARKS.
       THIS PROGRAM CREATES A FILE OF ALL USOCS WITH NETWORK
       ACCESS AND BUS. COMM. CLASS OF SERVICES (I.E. PBX, PABX,
       KTE, MAR, AND ERS).
            INPUT  - USOC RATE MASTER
            OUTPUT - NETWORK ACCESS USOC FILE
 MAINTENANCE HISTORY
 ==================
 V01   --- L LI --- 7010 --- NOV 20/86.
      NEW PROGRAM.
    SKIP1
 V21   --- B. LINDBERG --- GRA88 --- OCT 25/88.
      CHANGE URM FROM 14 TO 7 RATE GROUPS.
 ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
    SELECT            100-URM-FILE
      ASSIGN      TO  UT-S-URM
      FILE STATUS IS  1400-URM-STATUS.
    SELECT            200-USOC-FILE
      ASSIGN      TO  UT-S-USOC
      FILE STATUS IS  1400-USOC-STATUS.
 DATA DIVISION
 FILE SECTION.
004900 FD  100-URM-FILE
005000     LABEL RECORDS ARE STANDARD
005100     BLOCK CONTAINS 0 RECORDS
005200     DATA RECORD 100-URM-REC.
005300 01  100-URM-REC                  PIC  X(185).
005400     SKIP2
005500 FD  200-USOC-FILE
005600     LABEL RECORDS ARE STANDARD
005700     BLOCK CONTAINS 0 RECORDS
005800     DATA RECORD IS 200-USOC-REC.
005900 01  200-USOC-REC                 PIC  X(20).
006000     EJECT
006100 WORKING-STORAGE SECTION.
006200     SKIP1
006300 01  1000-URM-DETAIL.
006400     SKIP1
006500     05  1000-URM-USOC                PIC X(06).
006600         88  1000-URM-TRAILER           VALUE 'ÿÿÿÿÿÿ'.
006700*
006800*   THE ABOVE 88 LEVEL HAS THE HEX VALUE  'FFFFFFFFFFFF'
006900*
007000     05  1000-URM-CLASS               PIC X(01).
007100     05  1000-URM-COS-DESCRIPTION     PIC X(05).
007200     05  1000-URM-USOC-DESCRIPTION    PIC X(30).
007300     05  1000-URM-CURRENT-PRERATE-IND     PIC X(01).
007400     05  1000-URM-RATES OCCURS 7 TIMES.                     GRA88
007500         10  1000-USOC-BY-RG          PIC S9(03)V99 COMP-3.
007600     05      FILLER                PIC X(121).          GRA88
007700     SKIP2
007800 01  1100-MISC-FIELDS.
007900     SKIP1
008000     05  1100-LAST-USOC           PIC  X(06) VALUE SPACES.
008100     05  1100-CLASS-SERVICE       PIC  X(05).
008200         88  1100-BUS-COM-CLASS     VALUE 'PBX  '
008300                                    'PABX '
008400                                    'KTE  '
008500                                    'MAR  '
008600                                    'ERS  '.
008700     SKIP2
008800 01  1400-FILE-STATUS.
008900     SKIP1
009000     05  1400-URM-STATUS          PIC  X(02).
009100         88  1400-URM-STATUS-OK     VALUE '00' '10'.
```

```
009200        88  1400-URM-EOF           VALUE '10'.
009300     05  1400-USOC-STATUS           PIC  X(02).
009400        88  1400-USOC-STATUS-OK     VALUE '00'.
009500     SKIP2
009600 01  2000-USOC-DETAIL.
009700     SKIP1
009800     05  2000-USOC                  PIC  X(06).
009900     05  FILLER                   PIC  X(14) VALUE SPACES.
010000     SKIP2
010100 01  9900-ABEND-FIELD.
010200     SKIP1
010300     05  9900-STAT-OPCODE           PIC X(09) VALUE SPACES.
010400     05  9900-FILENAME              PIC X(08) VALUE SPACES.
010500     05  9900-PARANAME              PIC X(30) VALUE SPACES.
010600     05  9900-STAT-PROGNAME         PIC X(08) VALUE 'CR750'.
010700     05  9900-STAT-CODE             PIC X(02) VALUE SPACES.
010800     05  9900-STAT-ERROR-KEY        PIC X(50) VALUE SPACES.
010900     EJECT
011000 PROCEDURE DIVISION.
011100     SKIP1
011200 0000-CREATE-NETWORK-USOC-TABLE.
011300     SKIP1
011400     PERFORM 1000-INITIALIZE.
011500     SKIP1
011600     PERFORM  2000-PROCESS-URM
011700       UNTIL 1000-URM-TRAILER.
011800     SKIP1
011900     PERFORM 3000-TERMINATE.
012000     GOBACK.
012100     EJECT
012200 1000-INITIALIZE.
012300     SKIP2
012400     PERFORM 1100-OPEN-FILES.
012500     SKIP1
012600     PERFORM 9000-READ-URM.
012700     EJECT
012800 1100-OPEN-FILES.
012900     SKIP2
013000     OPEN INPUT  100-URM-FILE
013100          OUTPUT 200-USOC-FILE.
013200     SKIP2
013300     MOVE 'OPEN '  TO 9900-STAT-OPCODE
013400     MOVE '1100-INITILIZE' TO 9900-PARANAME
013500     SKIP1
013600     IF  1400-URM-STATUS-OK
013700     THEN
013800       NEXT SENTENCE
013900     ELSE
014000        MOVE 'URM '                TO 9900-FILENAME
014100        MOVE 1400-URM-STATUS        TO 9900-STAT-CODE
014200        CALL 'U076' USING 9900-ABEND-FIELD.
014300     SKIP2
014400     IF  1400-USOC-STATUS-OK
014500     THEN
014600       NEXT SENTENCE
014700     ELSE
014800        MOVE 'USOC'                TO 9900-FILENAME
014900        MOVE 1400-USOC-STATUS       TO 9900-STAT-CODE
015000        CALL 'U076' USING 9900-ABEND-FIELD.
015100     EJECT
015200 2000-PROCESS-URM.
015300     SKIP2
015400     MOVE  1000-URM-COS-DESCRIPTION TO 1100-CLASS-SERVICE.
015500     SKIP1
015600     IF    1000-USOC-BY-RG (1) NOT EQUAL TO  1000-USOC-BY-RG (2)  GRA88
015700      AND 1000-USOC-BY-RG (1) GREATER THAN  0                  GRA88
015800      AND 1000-URM-USOC       NOT EQUAL TO  1100-LAST-USOC
015900      AND 1100-BUS-COM-CLASS
016000     THEN
016100         MOVE 1000-URM-USOC  TO  1100-LAST-USOC
016200                                 2000-USOC
016300         PERFORM 9100-WRITE-USOC
016400     ELSE
016500        NEXT SENTENCE.
016600     SKIP1
016700     PERFORM 9000-READ-URM.
016800     EJECT
016900 3000-TERMINATE.
017000     SKIP2
```

```
017100     CLOSE 100-URM-FILE
017200           200-USOC-FILE.
017300     SKIP2
017400     MOVE 'CLOSE'                  TO 9900-STAT-OPCODE
017500     MOVE '3000-TERMINATE'         TO 9900-PARANAME
017600     SKIP1
017700     IF  1400-URM-STATUS-OK
017800     THEN
017900       NEXT SENTENCE
018000     ELSE
018100        MOVE 'URM '                TO 9900-FILENAME
018200        MOVE 1400-URM-STATUS       TO 9900-STAT-CODE
018300        CALL 'U076' USING 9900-ABEND-FIELD.
018400     SKIP2
018500     IF  1400-USOC-STATUS-OK
018600     THEN
018700       NEXT SENTENCE
018800     ELSE
018900        MOVE 'USOC'                TO 9900-FILENAME
019000        MOVE 1400-USOC-STATUS      TO 9900-STAT-CODE
019100        CALL 'U076' USING 9900-ABEND-FIELD.
019200     EJECT
019300 9000-READ-URM.
019400     SKIP2
019500     READ 100-URM-FILE INTO 1000-URM-DETAIL.
019600     SKIP1
019700     IF  1400-URM-STATUS-OK
019800     THEN
019900       NEXT SENTENCE
020000     ELSE
020100        MOVE 'READ '               TO 9900-STAT-OPCODE
020200        MOVE 'URM  '               TO 9900-FILENAME
020300        MOVE '9000-READ-URM'       TO 9900-PARANAME
020400        MOVE 1400-URM-STATUS       TO 9900-STAT-CODE
020500        CALL 'U076' USING 9900-ABEND-FIELD.
020600 9100-WRITE-USOC.
020700     WRITE 200-USOC-REC FROM 2000-USOC-DETAIL.
020800     SKIP1
020900     IF  1400-USOC-STATUS-OK
021000     THEN
021100         NEXT SENTENCE
021200     ELSE
021300        MOVE 'READ'                TO 9900-STAT-OPCODE
021400        MOVE 'USOC'                TO 9900-FILENAME
021500        MOVE '9100-WRITE-USOC'     TO 9900-PARANAME
021600        MOVE 1400-USOC-STATUS      TO 9900-STAT-CODE
021700        CALL 'U076' USING 9900-ABEND-FIELD.
```

Appendix B: List of WSL Code translated from COBOL Code

A. CR750V02.WSL

```
    /* IDENTIFICATION DIVISION. */

/* PROGRAM-ID.    CR750. */

/* AUTHOR.        LAWRENCE LI. */
/* INSTALLATION.  ALBERTA GOVERNMENT TELEPHONE. */

/* DATE-WRITTEN.  NOV 20/86. */

/* DATE-COMPILED. */

/* REMARKS. */
/*       THIS PROGRAM CREATES A FILE OF ALL USOCS WITH NETWORK */
/*       ACCESS AND BUS. COMM. CLASS OF SERVICES (I.E. PBX, PABX, */
/*       KTE, MAR, AND ERS). */
/*             INPUT  - USOC RATE MASTER */

/*             OUTPUT - NETWORK ACCESS USOC FILE */

/* MAINTENANCE HISTORY */
/*  ================== */

/* V01   --- L LI --- 7010 --- NOV 20/86. */
/*       NEW PROGRAM. */

/* V21   --- B. LINDBERG --- GRA88 --- OCT 25/88. */
/*       CHANGE URM FROM 14 TO 7 RATE GROUPS. */

/* ENVIRONMENT DIVISION. */

/* INPUT-OUTPUT SECTION. */

 /* FILE-CONTROL. */


VAR  UT-S-URM

VAR  UT-S-USOC

/*  DATA DIVISION */
/* FILE SECTION. */

  VAR STRUCT 100-URM-FILE
   BEGIN
    VAR STRUCT ELEMENT1
     BEGIN
      VAR  ELEMENT1
     END
    VAR ELEMENT2
    VAR ELEMENT3
    VAR ELEMENT4
    VAR ELEMENT5
    VAR STRUCT ELEMENT6
     BEGIN
       VAR ELEMENT1
       VAR ELEMENT2
       VAR ELEMENT3
       VAR ELEMENT4
       VAR ELEMENT5
       VAR ELEMENT6
       VAR ELEMENT7
     END
    VAR ELEMENT7
    VAR INDEXELEMENT1
    VAR INDEXELEMENT2
    VAR INDEXELEMENT3
    VAR INDEXELEMENT4
    VAR INDEXELEMENT5
    VAR STRUCT INDEXELEMENT6
     BEGIN
```

```
    VAR INDEXELEMENT1
    VAR INDEXELEMENT2
    VAR INDEXELEMENT3
    VAR INDEXELEMENT4
    VAR INDEXELEMENT5
    VAR INDEXELEMENT6
    VAR INDEXELEMENT7
   END
  VAR INDEXELEMENT7
 END

 VAR   100-URM-REC
 /* VAR 100-URM-REC       */

 VAR  200-USOC-FILE
 VAR STRUCT 200-USOC-REC
  BEGIN
   VAR ELEMENT1
   VAR ELEMENT2
  END



/*  WORKING-STORAGE SECTION. */

 VAR STRUCT 1000-URM-DETAIL
  BEGIN
    VAR STRUCT 1000-URM-USOC
     BEGIN
       VAR  1000-URM-TRAILER
     END
    VAR 1000-URM-CLASS
    VAR  1000-URM-COS-DESCRIPTION
    VAR  1000-URM-USOC-DESCRIPTION
    VAR  1000-URM-CURRENT-PRERATE-IND
    VAR  STRUCT 1000-URM-RATES[7]
     BEGIN
       VAR  1000-USOC-BY-RG
     END
    VAR     FILLER1
 END

 VAR STRUCT  1100-MISC-FIELDS
  BEGIN
    VAR  1100-LAST-USOC
    VAR STRUCT  1100-CLASS-SERVICE
     BEGIN
       VAR 1100-BUS-COM-CLASS
     END
  END

 VAR STRUCT 1400-FILE-STATUS
  BEGIN
    VAR STRUCT  1400-URM-STATUS
     BEGIN
       VAR  1400-URM-STATUS-OK
       VAR  1400-URM-EOF
     END
    VAR STRUCT  1400-USOC-STATUS
     BEGIN
       VAR  1400-USOC-STATUS-OK
     END
 END

 VAR STRUCT 2000-USOC-DETAIL
  BEGIN
    VAR 2000-USOC
    VAR FILLER1
    VAR INDEXELEMENT1
    VAR INDEXELEMENT2
  END

VAR STRUCT  9900-ABEND-FIELD
 BEGIN
    VAR 9900-STAT-OPCODE
    VAR 9900-FILENAME
    VAR 9900-PARANAME
    VAR 9900-STAT-PROGNAME
    VAR 9900-STAT-CODE
```

```
    VAR 9900-STAT-ERROR-KEY
 END

/* PROCEDURE DIVISION. */

 PROC CR705V02_VAR-INIT( 1100-MISC-FIELDs.1100-CLASS-SERVICE , 1000-URM-DETAIL.1000-URM-COS-DESCRIPTION ,
1000-URM-DETAIL.1000-URM-RATES[1].1000-USOC-BY-RG , 1000-URM-DETAIL.1000-URM-RATES[2].1000-USOC-BY-RG ,
1000-URM-DETAIL.1000-URM-RATES[1].1000-USOC-BY-RG , 1000-URM-DETAIL.1000-URM-USOC ,
1100-MISC-FIELDS.1100-LAST-USOC , 1100-MISC-FIELDS.1100-CLASS-SERVICE.1100-BUS-COM-CLASS ,
1100-MISC-FIELDS.1100-LAST-USOC , 1000-URM-DETAIL.1000-URM-USOC , 2000-USOC-DETAIL.2000-USOC ,
1100-MISC-FIELDS.1100-LAST-USOC , 1000-URM-DETAIL.1000-URM-USOC.1000-URM-TRAILER ,
100-URM-FILE.ELEMENT1.ELEMENT1, 100-URM-FILE.INDEXELEMENT1.INDEXELEMENT1 ,
1000-URM-DETAIL.1000-URM-CLASS , 100-URM-FILE.ELEMENT2 , 100-URM-FILE.INDEXELEMENT2 ,
1000-URM-DETAIL.1000-URM-COS-DESCRIPTION , 100-URM-FILE.ELEMENT3 , 100-URM-FILE.INDEXELEMENT3 ,
1000-URM-DETAIL.1000-URM-USOC-DESCRIPTION , 100-URM-FILE.ELEMENT4 , 100-URM-FILE.INDEXELEMENT4 ,
1000-URM-DETAIL.1000-URM-CURRENT-PRERATE-IND , 100-URM-FILE.ELEMENT5 , 100-URM-FILE.INDEXELEMENT5 ,
1000-URM-DETAIL.1000-URM-RATES[1].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT1 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT1, 1000-URM-DETAIL.1000-URM-RATES[2].1000-USOC-BY-RG ,
100-URM-FILE.ELEMENT6.ELEMENT2 , 100-URM-FILE.INDEXELEMENT6.INDEXELEMENT2 ,
1000-URM-DETAIL.1000-URM-RATES[3].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT3 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT3 , 1000-URM-DETAIL.1000-URM-RATES[4].1000-USOC-BY-RG ,
100-URM-FILE.ELEMENT6.ELEMENT4 , 100-URM-FILE.INDEXELEMENT6.INDEXELEMENT4 ,
1000-URM-DETAIL.1000-URM-RATES[5].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT5 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT5 , 1000-URM-DETAIL.1000-URM-RATES[6].1000-USOC-BY-RG ,
100-URM-FILE.ELEMENT6.ELEMENT6 , 100-URM-FILE.INDEXELEMENT6.INDEXELEMENT6 ,
1000-URM-DETAIL.1000-URM-RATES[7].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT7 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT7 , 1000-URM-DETAIL.FILLER1 , 100-URM-FILE.ELEMENT7 ,
100-URM-FILE.INDEXELEMENT7, 1400-FILE-STATUS.1400-URM-STATUS.1400-URM-STATUS-OK ,
9900-ABEND-FIELD.9900-STAT-OPCODE ,  9900-ABEND-FIELD.9900-FILENAME , 9900-ABEND-FIELD.9900-PARANAME ,
9900-ABEND-FIELD.9900-STAT-CODE , 9900-ABEND-FIELD , 200-USOC-REC.ELEMENT1 , 2000-USOC-DETAIL.2000-USOC ,
2000-USOC-DETAIL.INDEXELEMENT1,200-USOC-REC.ELEMENT2 , 2000-USOC-DETAIL.FILLER1 ,
2000-USOC-DETAIL.INDEXELEMENT2 , 1400-FILE-STATUS.1400-USOC-STATUS.1400-USOC-STATUS-OK ,
9900-ABEND-FIELD.9900-STAT-OPCODE , 9900-ABEND-FIELD.9900-PARANAME , 9900-ABEND-FIELD.9900-STAT-CODE ,
9900-ABEND-FIELD , 100-URM-FILE , 200-USOC-FILE , 9900-ABEND-FIELD.9900-STAT-OPCODE ,
9900-ABEND-FIELD.9900-PARANAME, 1400-FILE-STATUS.1400-URM-STATUS.1400-URM-STATUS-OK ,
9900-ABEND-FIELD.9900-FILENAME , 9900-ABEND-FIELD.9900-STAT-CODE , 9900-ABEND-FIELD ,
1400-FILE-STATUS.1400-USOC-STATUS.1400-USOC-STATUS-OK, 1000-URM-DETAIL.1000-URM-USOC.1000-URM-TRAILER ,
100-URM-FILE.ELEMENT1.ELEMENT1, 100-URM-FILE.INDEXELEMENT1.INDEXELEMENT1 ,
1000-URM-DETAIL.1000-URM-CLASS , 100-URM-FILE.ELEMENT2 , 100-URM-FILE.INDEXELEMENT2 ,
1000-URM-DETAIL.1000-URM-COS-DESCRIPTION , 100-URM-FILE.ELEMENT3 , 100-URM-FILE.INDEXELEMENT3 ,
1000-URM-DETAIL.1000-URM-USOC-DESCRIPTION , 100-URM-FILE.ELEMENT4 , 100-URM-FILE.INDEXELEMENT4 ,
1000-URM-DETAIL.1000-URM-CURRENT-PRERATE-IND , 100-URM-FILE.ELEMENT5 , 100-URM-FILE.INDEXELEMENT5 ,
1000-URM-DETAIL.1000-URM-RATES[1].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT1 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT1, 1000-URM-DETAIL.1000-URM-RATES[2].1000-USOC-BY-RG ,
100-URM-FILE.ELEMENT6.ELEMENT2 , 100-URM-FILE.INDEXELEMENT6.INDEXELEMENT2 ,
1000-URM-DETAIL.1000-URM-RATES[3].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT3 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT3 , 1000-URM-DETAIL.1000-URM-RATES[4].1000-USOC-BY-RG ,
100-URM-FILE.ELEMENT6.ELEMENT4 , 100-URM-FILE.INDEXELEMENT6.INDEXELEMENT4 ,
1000-URM-DETAIL.1000-URM-RATES[5].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT5 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT5 , 1000-URM-DETAIL.1000-URM-RATES[6].1000-USOC-BY-RG ,
100-URM-FILE.ELEMENT6.ELEMENT6 , 100-URM-FILE.INDEXELEMENT6.INDEXELEMENT6 ,
1000-URM-DETAIL.1000-URM-RATES[7].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT7 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT7 , 1000-URM-DETAIL.FILLER1 , 100-URM-FILE.ELEMENT7 ,
100-URM-FILE.INDEXELEMENT7, 1400-FILE-STATUS.1400-URM-STATUS.1400-URM-STATUS-OK ,
9900-ABEND-FIELD.9900-STAT-OPCODE ,  9900-ABEND-FIELD.9900-FILENAME , 9900-ABEND-FIELD.9900-PARANAME ,
9900-ABEND-FIELD.9900-STAT-CODE)
   BEGIN
    100-URM-FILE.PHYS_DEV := 'UT-S-URM'
    100-URM-FILE.DATARECORD := 100-URM-REC
    200-USOC-FILE.PHYS_DEV := 'UT-S-USOC'
    200-USOC-FILE.DATARECORD := 200-USOC-REC
   9900-ABEND-FIELD.9900-PARANAME := ' '
   9900-ABEND-FIELD.9900-STAT-PROGNAME  := 'CR750'
   1400-FILE-STATUS.1400-URM-EOF.1400-URM-STATUS-OK := '10'
   1400-FILE-STATUS.1400-URM-EOF.1400-URM-EOF := '10'
   1000-URM-DETAIL.1000-URM-USOC.1000-URM-TRAILER := 'ÿÿÿÿÿÿ'
   CALL 0000-CREATE-NETWORK-USOC-TABLE( 1100-MISC-FIELDs.1100-CLASS-SERVICE ,
1000-URM-DETAIL.1000-URM-COS-DESCRIPTION , 1000-URM-DETAIL.1000-URM-RATES[1].1000-USOC-BY-RG ,
1000-URM-DETAIL.1000-URM-RATES[2].1000-USOC-BY-RG , 1000-URM-DETAIL.1000-URM-RATES[1].1000-USOC-BY-RG ,
1000-URM-DETAIL.1000-URM-USOC , 1100-MISC-FIELDS.1100-LAST-USOC ,
1100-MISC-FIELDS.1100-CLASS-SERVICE.1100-BUS-COM-CLASS , 1100-MISC-FIELDS.1100-LAST-USOC ,
1000-URM-DETAIL.1000-URM-USOC , 2000-USOC-DETAIL.2000-USOC , 1100-MISC-FIELDS.1100-LAST-USOC ,
1000-URM-DETAIL.1000-URM-USOC.1000-URM-TRAILER , 100-URM-FILE.ELEMENT1.ELEMENT1,
100-URM-FILE.INDEXELEMENT1.INDEXELEMENT1 , 1000-URM-DETAIL.1000-URM-CLASS , 100-URM-FILE.ELEMENT2 ,
100-URM-FILE.INDEXELEMENT2 , 1000-URM-DETAIL.1000-URM-COS-DESCRIPTION , 100-URM-FILE.ELEMENT3 ,
100-URM-FILE.INDEXELEMENT3 , 1000-URM-DETAIL.1000-URM-USOC-DESCRIPTION , 100-URM-FILE.ELEMENT4 ,
100-URM-FILE.INDEXELEMENT4 , 1000-URM-DETAIL.1000-URM-CURRENT-PRERATE-IND , 100-URM-FILE.ELEMENT5 ,
100-URM-FILE.INDEXELEMENT5 , 1000-URM-DETAIL.1000-URM-RATES[1].1000-USOC-BY-RG ,
100-URM-FILE.ELEMENT6.ELEMENT1 , 100-URM-FILE.INDEXELEMENT6.INDEXELEMENT1,
```

1000-URM-DETAIL.1000-URM-RATES[2].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT2 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT2 , 1000-URM-DETAIL.1000-URM-RATES[3].1000-USOC-BY-RG ,
100-URM-FILE.ELEMENT6.ELEMENT3 , 100-URM-FILE.INDEXELEMENT6.INDEXELEMENT3 ,
1000-URM-DETAIL.1000-URM-RATES[4].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT4 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT4 , 1000-URM-DETAIL.1000-URM-RATES[5].1000-USOC-BY-RG ,
100-URM-FILE.ELEMENT6.ELEMENT5 , 100-URM-FILE.INDEXELEMENT6.INDEXELEMENT5 ,
1000-URM-DETAIL.1000-URM-RATES[6].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT6 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT6 , 1000-URM-DETAIL.1000-URM-RATES[7].1000-USOC-BY-RG ,
100-URM-FILE.ELEMENT6.ELEMENT7 , 100-URM-FILE.INDEXELEMENT6.INDEXELEMENT7 , 1000-URM-DETAIL.FILLER1 ,
100-URM-FILE.ELEMENT7 , 100-URM-FILE.INDEXELEMENT7, 1400-FILE-STATUS.1400-URM-STATUS.1400-URM-STATUS-OK ,
9900-ABEND-FIELD.9900-STAT-OPCODE , 9900-ABEND-FIELD.9900-FILENAME , 9900-ABEND-FIELD.9900-PARANAME ,
9900-ABEND-FIELD.9900-STAT-CODE , 9900-ABEND-FIELD , 200-USOC-REC.ELEMENT1 , 2000-USOC-DETAIL.2000-USOC ,
2000-USOC-DETAIL.INDEXELEMENT1,200-USOC-REC.ELEMENT2 , 2000-USOC-DETAIL.FILLER1 ,
2000-USOC-DETAIL.INDEXELEMENT2 , 1400-FILE-STATUS.1400-USOC-STATUS.1400-USOC-STATUS-OK ,
9900-ABEND-FIELD.9900-STAT-OPCODE , 9900-ABEND-FIELD.9900-PARANAME , 9900-ABEND-FIELD.9900-STAT-CODE ,
9900-ABEND-FIELD , 100-URM-FILE , 200-USOC-FILE , 9900-ABEND-FIELD.9900-STAT-OPCODE ,
9900-ABEND-FIELD.9900-PARANAME, 1400-FILE-STATUS.1400-URM-STATUS.1400-URM-STATUS-OK ,
9900-ABEND-FIELD.9900-FILENAME , 9900-ABEND-FIELD.9900-STAT-CODE , 9900-ABEND-FIELD ,
1400-FILE-STATUS.1400-USOC-STATUS.1400-USOC-STATUS-OK, 1000-URM-DETAIL.1000-URM-USOC.1000-URM-TRAILER ,
100-URM-FILE.ELEMENT1.ELEMENT1, 100-URM-FILE.INDEXELEMENT1.INDEXELEMENT1 ,
1000-URM-DETAIL.1000-URM-CLASS , 100-URM-FILE.ELEMENT2 , 100-URM-FILE.INDEXELEMENT2 ,
1000-URM-DETAIL.1000-URM-COS-DESCRIPTION , 100-URM-FILE.ELEMENT3 , 100-URM-FILE.INDEXELEMENT3 ,
1000-URM-DETAIL.1000-URM-USOC-DESCRIPTION , 100-URM-FILE.ELEMENT4 , 100-URM-FILE.INDEXELEMENT4 ,
1000-URM-DETAIL.1000-URM-CURRENT-PRERATE-IND , 100-URM-FILE.ELEMENT5 , 100-URM-FILE.INDEXELEMENT5 ,
1000-URM-DETAIL.1000-URM-RATES[1].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT1 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT1, 1000-URM-DETAIL.1000-URM-RATES[2].1000-USOC-BY-RG ,
100-URM-FILE.ELEMENT6.ELEMENT2 , 100-URM-FILE.INDEXELEMENT6.INDEXELEMENT2 ,
1000-URM-DETAIL.1000-URM-RATES[3].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT3 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT3 , 1000-URM-DETAIL.1000-URM-RATES[4].1000-USOC-BY-RG ,
100-URM-FILE.ELEMENT6.ELEMENT4 , 100-URM-FILE.INDEXELEMENT6.INDEXELEMENT4 ,
1000-URM-DETAIL.1000-URM-RATES[5].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT5 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT5 , 1000-URM-DETAIL.1000-URM-RATES[6].1000-USOC-BY-RG ,
100-URM-FILE.ELEMENT6.ELEMENT6 , 100-URM-FILE.INDEXELEMENT6.INDEXELEMENT6 ,
1000-URM-DETAIL.1000-URM-RATES[7].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT7 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT7 , 1000-URM-DETAIL.FILLER1 , 100-URM-FILE.ELEMENT7 ,
100-URM-FILE.INDEXELEMENT7, 1400-FILE-STATUS.1400-URM-STATUS.1400-URM-STATUS-OK ,
9900-ABEND-FIELD.9900-STAT-OPCODE , 9900-ABEND-FIELD.9900-FILENAME , 9900-ABEND-FIELD.9900-PARANAME ,
9900-ABEND-FIELD.9900-STAT-CODE)
END.

PROC 0000-CREATE-NETWORK-USOC-TABLE(1100-MISC-FIELDs.1100-CLASS-SERVICE ,
1000-URM-DETAIL.1000-URM-COS-DESCRIPTION , 1000-URM-DETAIL.1000-URM-RATES[1].1000-USOC-BY-RG ,
1000-URM-DETAIL.1000-URM-RATES[2].1000-USOC-BY-RG , 1000-URM-DETAIL.1000-URM-RATES[1].1000-USOC-BY-RG ,
1000-URM-DETAIL.1000-URM-USOC , 1100-MISC-FIELDS.1100-LAST-USOC ,
1100-MISC-FIELDS.1100-CLASS-SERVICE.1100-BUS-COM-CLASS , 1100-MISC-FIELDS.1100-LAST-USOC ,
1000-URM-DETAIL.1000-URM-USOC , 2000-USOC-DETAIL.2000-USOC , 1100-MISC-FIELDS.1100-LAST-USOC ,
1000-URM-DETAIL.1000-URM-USOC.1000-URM-TRAILER , 100-URM-FILE.ELEMENT1.ELEMENT1,
100-URM-FILE.INDEXELEMENT1.INDEXELEMENT1 , 1000-URM-DETAIL.1000-URM-CLASS , 100-URM-FILE.ELEMENT2 ,
100-URM-FILE.INDEXELEMENT2 , 1000-URM-DETAIL.1000-URM-COS-DESCRIPTION , 100-URM-FILE.ELEMENT3 ,
100-URM-FILE.INDEXELEMENT3 , 1000-URM-DETAIL.1000-URM-USOC-DESCRIPTION , 100-URM-FILE.ELEMENT4 ,
100-URM-FILE.INDEXELEMENT4 , 1000-URM-DETAIL.1000-URM-CURRENT-PRERATE-IND , 100-URM-FILE.ELEMENT5 ,
100-URM-FILE.INDEXELEMENT5 , 1000-URM-DETAIL.1000-URM-RATES[1].1000-USOC-BY-RG ,
100-URM-FILE.ELEMENT6.ELEMENT1 , 100-URM-FILE.INDEXELEMENT6.INDEXELEMENT1,
1000-URM-DETAIL.1000-URM-RATES[2].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT2 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT2 , 1000-URM-DETAIL.1000-URM-RATES[3].1000-USOC-BY-RG ,
100-URM-FILE.ELEMENT6.ELEMENT3 , 100-URM-FILE.INDEXELEMENT6.INDEXELEMENT3 ,
1000-URM-DETAIL.1000-URM-RATES[4].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT4 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT4 , 1000-URM-DETAIL.1000-URM-RATES[5].1000-USOC-BY-RG ,
100-URM-FILE.ELEMENT6.ELEMENT5 , 100-URM-FILE.INDEXELEMENT6.INDEXELEMENT5 ,
1000-URM-DETAIL.1000-URM-RATES[6].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT6 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT6 , 1000-URM-DETAIL.1000-URM-RATES[7].1000-USOC-BY-RG ,
100-URM-FILE.ELEMENT6.ELEMENT7 , 100-URM-FILE.INDEXELEMENT6.INDEXELEMENT7 , 1000-URM-DETAIL.FILLER1 ,
100-URM-FILE.ELEMENT7 , 100-URM-FILE.INDEXELEMENT7, 1400-FILE-STATUS.1400-URM-STATUS.1400-URM-STATUS-OK ,
9900-ABEND-FIELD.9900-STAT-OPCODE , 9900-ABEND-FIELD.9900-FILENAME , 9900-ABEND-FIELD.9900-PARANAME ,
9900-ABEND-FIELD.9900-STAT-CODE , 9900-ABEND-FIELD , 200-USOC-REC.ELEMENT1 , 2000-USOC-DETAIL.2000-USOC ,
2000-USOC-DETAIL.INDEXELEMENT1,200-USOC-REC.ELEMENT2 , 2000-USOC-DETAIL.FILLER1 ,
2000-USOC-DETAIL.INDEXELEMENT2 , 1400-FILE-STATUS.1400-USOC-STATUS.1400-USOC-STATUS-OK ,
9900-ABEND-FIELD.9900-STAT-OPCODE , 9900-ABEND-FIELD.9900-PARANAME , 9900-ABEND-FIELD.9900-STAT-CODE ,
9900-ABEND-FIELD , 100-URM-FILE , 200-USOC-FILE , 9900-ABEND-FIELD.9900-STAT-OPCODE ,
9900-ABEND-FIELD.9900-PARANAME, 1400-FILE-STATUS.1400-URM-STATUS.1400-URM-STATUS-OK ,
9900-ABEND-FIELD.9900-FILENAME , 9900-ABEND-FIELD.9900-STAT-CODE , 9900-ABEND-FIELD ,
1400-FILE-STATUS.1400-USOC-STATUS.1400-USOC-STATUS-OK, 1000-URM-DETAIL.1000-URM-USOC.1000-URM-TRAILER ,
100-URM-FILE.ELEMENT1.ELEMENT1, 100-URM-FILE.INDEXELEMENT1.INDEXELEMENT1 ,
1000-URM-DETAIL.1000-URM-CLASS , 100-URM-FILE.ELEMENT2 , 100-URM-FILE.INDEXELEMENT2 ,
1000-URM-DETAIL.1000-URM-COS-DESCRIPTION , 100-URM-FILE.ELEMENT3 , 100-URM-FILE.INDEXELEMENT3 ,
1000-URM-DETAIL.1000-URM-USOC-DESCRIPTION , 100-URM-FILE.ELEMENT4 , 100-URM-FILE.INDEXELEMENT4 ,
1000-URM-DETAIL.1000-URM-CURRENT-PRERATE-IND , 100-URM-FILE.ELEMENT5 , 100-URM-FILE.INDEXELEMENT5 ,
1000-URM-DETAIL.1000-URM-RATES[1].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT1 ,

```
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT1, 1000-URM-DETAIL.1000-URM-RATES[2].1000-USOC-BY-RG ,
100-URM-FILE.ELEMENT6.ELEMENT2 , 100-URM-FILE.INDEXELEMENT6.INDEXELEMENT2 ,
1000-URM-DETAIL.1000-URM-RATES[3].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT3 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT3 , 1000-URM-DETAIL.1000-URM-RATES[4].1000-USOC-BY-RG ,
100-URM-FILE.ELEMENT6.ELEMENT4 , 100-URM-FILE.INDEXELEMENT6.INDEXELEMENT4 ,
1000-URM-DETAIL.1000-URM-RATES[5].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT5 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT5 , 1000-URM-DETAIL.1000-URM-RATES[6].1000-USOC-BY-RG ,
100-URM-FILE.ELEMENT6.ELEMENT6 , 100-URM-FILE.INDEXELEMENT6.INDEXELEMENT6 ,
1000-URM-DETAIL.1000-URM-RATES[7].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT7 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT7 , 1000-URM-DETAIL.FILLER1 , 100-URM-FILE.ELEMENT7 ,
100-URM-FILE.INDEXELEMENT7, 1400-FILE-STATUS.1400-URM-STATUS.1400-URM-STATUS-OK ,
9900-ABEND-FIELD.9900-STAT-OPCODE ,  9900-ABEND-FIELD.9900-FILENAME , 9900-ABEND-FIELD.9900-PARANAME ,
9900-ABEND-FIELD.9900-STAT-CODE  )
  BEGIN
   CALL 1000-INITIALIZE( 100-URM-FILE , 200-USOC-FILE , 9900-ABEND-FIELD.9900-STAT-OPCODE ,
9900-ABEND-FIELD.9900-PARANAME, 1400-FILE-STATUS.1400-URM-STATUS.1400-URM-STATUS-OK ,
9900-ABEND-FIELD.9900-FILENAME , 9900-ABEND-FIELD.9900-STAT-CODE , 9900-ABEND-FIELD ,
1400-FILE-STATUS.1400-USOC-STATUS.1400-USOC-STATUS-OK, 1000-URM-DETAIL.1000-URM-USOC.1000-URM-TRAILER ,
100-URM-FILE.ELEMENT1.ELEMENT1, 100-URM-FILE.INDEXELEMENT1.INDEXELEMENT1 ,
1000-URM-DETAIL.1000-URM-CLASS , 100-URM-FILE.ELEMENT2 , 100-URM-FILE.INDEXELEMENT2 ,
1000-URM-DETAIL.1000-URM-COS-DESCRIPTION , 100-URM-FILE.ELEMENT3 , 100-URM-FILE.INDEXELEMENT3 ,
1000-URM-DETAIL.1000-URM-USOC-DESCRIPTION , 100-URM-FILE.ELEMENT4 , 100-URM-FILE.INDEXELEMENT4 ,
1000-URM-DETAIL.1000-URM-CURRENT-PRERATE-IND , 100-URM-FILE.ELEMENT5 , 100-URM-FILE.INDEXELEMENT5 ,
1000-URM-DETAIL.1000-URM-RATES[1].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT1 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT1, 1000-URM-DETAIL.1000-URM-RATES[2].1000-USOC-BY-RG ,
100-URM-FILE.ELEMENT6.ELEMENT2 , 100-URM-FILE.INDEXELEMENT6.INDEXELEMENT2 ,
1000-URM-DETAIL.1000-URM-RATES[3].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT3 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT3 , 1000-URM-DETAIL.1000-URM-RATES[4].1000-USOC-BY-RG ,
100-URM-FILE.ELEMENT6.ELEMENT4 , 100-URM-FILE.INDEXELEMENT6.INDEXELEMENT4 ,
1000-URM-DETAIL.1000-URM-RATES[5].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT5 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT5 , 1000-URM-DETAIL.1000-URM-RATES[6].1000-USOC-BY-RG ,
100-URM-FILE.ELEMENT6.ELEMENT6 , 100-URM-FILE.INDEXELEMENT6.INDEXELEMENT6 ,
1000-URM-DETAIL.1000-URM-RATES[7].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT7 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT7 , 1000-URM-DETAIL.FILLER1 , 100-URM-FILE.ELEMENT7 ,
100-URM-FILE.INDEXELEMENT7, 1400-FILE-STATUS.1400-URM-STATUS.1400-URM-STATUS-OK ,
9900-ABEND-FIELD.9900-STAT-OPCODE ,  9900-ABEND-FIELD.9900-FILENAME , 9900-ABEND-FIELD.9900-PARANAME ,
9900-ABEND-FIELD.9900-STAT-CODE  )
   WHILE NOT 1000-URM-DETAIL.1000-URM-USOC.1000-URM-TRAILER DO
   CALL 2000-PROCESS-URM( 1100-MISC-FIELDs.1100-CLASS-SERVICE , 1000-URM-DETAIL.1000-URM-COS-DESCRIPTION ,
1000-URM-DETAIL.1000-URM-RATES[1].1000-USOC-BY-RG , 1000-URM-DETAIL.1000-URM-RATES[2].1000-USOC-BY-RG ,
1000-URM-DETAIL.1000-URM-RATES[1].1000-USOC-BY-RG , 1000-URM-DETAIL.1000-URM-USOC ,
1100-MISC-FIELDS.1100-LAST-USOC , 1100-MISC-FIELDS.1100-CLASS-SERVICE.1100-BUS-COM-CLASS ,
1100-MISC-FIELDS.1100-LAST-USOC , 1000-URM-DETAIL.1000-URM-USOC , 2000-USOC-DETAIL.2000-USOC ,
1100-MISC-FIELDS.1100-LAST-USOC , 1000-URM-DETAIL.1000-URM-USOC.1000-URM-TRAILER ,
100-URM-FILE.ELEMENT1.ELEMENT1, 100-URM-FILE.INDEXELEMENT1.INDEXELEMENT1 ,
1000-URM-DETAIL.1000-URM-CLASS , 100-URM-FILE.ELEMENT2 , 100-URM-FILE.INDEXELEMENT2 ,
1000-URM-DETAIL.1000-URM-COS-DESCRIPTION , 100-URM-FILE.ELEMENT3 , 100-URM-FILE.INDEXELEMENT3 ,
1000-URM-DETAIL.1000-URM-USOC-DESCRIPTION , 100-URM-FILE.ELEMENT4 , 100-URM-FILE.INDEXELEMENT4 ,
1000-URM-DETAIL.1000-URM-CURRENT-PRERATE-IND , 100-URM-FILE.ELEMENT5 , 100-URM-FILE.INDEXELEMENT5 ,
1000-URM-DETAIL.1000-URM-RATES[1].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT1 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT1, 1000-URM-DETAIL.1000-URM-RATES[2].1000-USOC-BY-RG ,
100-URM-FILE.ELEMENT6.ELEMENT2 , 100-URM-FILE.INDEXELEMENT6.INDEXELEMENT2 ,
1000-URM-DETAIL.1000-URM-RATES[3].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT3 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT3 , 1000-URM-DETAIL.1000-URM-RATES[4].1000-USOC-BY-RG ,
100-URM-FILE.ELEMENT6.ELEMENT4 , 100-URM-FILE.INDEXELEMENT6.INDEXELEMENT4 ,
1000-URM-DETAIL.1000-URM-RATES[5].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT5 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT5 , 1000-URM-DETAIL.1000-URM-RATES[6].1000-USOC-BY-RG ,
100-URM-FILE.ELEMENT6.ELEMENT6 , 100-URM-FILE.INDEXELEMENT6.INDEXELEMENT6 ,
1000-URM-DETAIL.1000-URM-RATES[7].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT7 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT7 , 1000-URM-DETAIL.FILLER1 , 100-URM-FILE.ELEMENT7 ,
100-URM-FILE.INDEXELEMENT7, 1400-FILE-STATUS.1400-URM-STATUS.1400-URM-STATUS-OK ,
9900-ABEND-FIELD.9900-STAT-OPCODE ,  9900-ABEND-FIELD.9900-FILENAME , 9900-ABEND-FIELD.9900-PARANAME ,
9900-ABEND-FIELD.9900-STAT-CODE , 9900-ABEND-FIELD , 200-USOC-REC.ELEMENT1 , 2000-USOC-DETAIL.2000-USOC ,
2000-USOC-DETAIL.INDEXELEMENT1,200-USOC-REC.ELEMENT2 , 2000-USOC-DETAIL.FILLER1 ,
2000-USOC-DETAIL.INDEXELEMENT2 , 1400-FILE-STATUS.1400-USOC-STATUS.1400-USOC-STATUS-OK ,
9900-ABEND-FIELD.9900-STAT-OPCODE , 9900-ABEND-FIELD.9900-PARANAME , 9900-ABEND-FIELD.9900-STAT-CODE ,
9900-ABEND-FIELD  )
   OD
   CALL 3000-TERMINATE( 100-URM-FILE , 200-USOC-FILE , 9900-ABEND-FIELDS.9900-STAT-OPCODE ,
9900-ABEND-FIELDS.9900-PARANAME , 1400-FILE-STATUS.1400-URM-STATUS.1400-URM-STATUS-OK ,
9900-ABEND-FIELDS.9900-FILENAME , 9900-ABEND-FIELDS.9900-STAT-CODE , 9900-ABEND-FIELD ,
1400-FILE-STATUS.1400-USOC-STATUS.1400-USOC-STATUS-OK  )
  END.

 PROC  1000-INITIALIZE( 100-URM-FILE , 200-USOC-FILE , 9900-ABEND-FIELD.9900-STAT-OPCODE ,
9900-ABEND-FIELD.9900-PARANAME, 1400-FILE-STATUS.1400-URM-STATUS.1400-URM-STATUS-OK ,
9900-ABEND-FIELD.9900-FILENAME , 9900-ABEND-FIELD.9900-STAT-CODE , 9900-ABEND-FIELD ,
1400-FILE-STATUS.1400-USOC-STATUS.1400-USOC-STATUS-OK, 1000-URM-DETAIL.1000-URM-USOC.1000-URM-TRAILER ,
100-URM-FILE.ELEMENT1.ELEMENT1, 100-URM-FILE.INDEXELEMENT1.INDEXELEMENT1 ,
```

```
1000-URM-DETAIL.1000-URM-CLASS , 100-URM-FILE.ELEMENT2 , 100-URM-FILE.INDEXELEMENT2 ,
1000-URM-DETAIL.1000-URM-COS-DESCRIPTION , 100-URM-FILE.ELEMENT3 , 100-URM-FILE.INDEXELEMENT3 ,
1000-URM-DETAIL.1000-URM-USOC-DESCRIPTION , 100-URM-FILE.ELEMENT4 , 100-URM-FILE.INDEXELEMENT4 ,
1000-URM-DETAIL.1000-URM-CURRENT-PRERATE-IND , 100-URM-FILE.ELEMENT5 , 100-URM-FILE.INDEXELEMENT5 ,
1000-URM-DETAIL.1000-URM-RATES[1].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT1 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT1, 1000-URM-DETAIL.1000-URM-RATES[2].1000-USOC-BY-RG ,
100-URM-FILE.ELEMENT6.ELEMENT2 , 100-URM-FILE.INDEXELEMENT6.INDEXELEMENT2 ,
1000-URM-DETAIL.1000-URM-RATES[3].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT3 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT3 , 1000-URM-DETAIL.1000-URM-RATES[4].1000-USOC-BY-RG ,
100-URM-FILE.ELEMENT6.ELEMENT4 , 100-URM-FILE.INDEXELEMENT6.INDEXELEMENT4 ,
1000-URM-DETAIL.1000-URM-RATES[5].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT5 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT5 , 1000-URM-DETAIL.1000-URM-RATES[6].1000-USOC-BY-RG ,
100-URM-FILE.ELEMENT6.ELEMENT6 , 100-URM-FILE.INDEXELEMENT6.INDEXELEMENT6 ,
1000-URM-DETAIL.1000-URM-RATES[7].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT7 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT7 , 1000-URM-DETAIL.FILLER1 , 100-URM-FILE.ELEMENT7 ,
100-URM-FILE.INDEXELEMENT7, 1400-FILE-STATUS.1400-URM-STATUS.1400-URM-STATUS-OK ,
9900-ABEND-FIELD.9900-STAT-OPCODE ,  9900-ABEND-FIELD.9900-FILENAME , 9900-ABEND-FIELD.9900-PARANAME ,
9900-ABEND-FIELD.9900-STAT-CODE   )
  BEGIN
    CALL 1100-OPEN-FILES( 100-URM-FILE , 200-USOC-FILE , 9900-ABEND-FIELD.9900-STAT-OPCODE ,
9900-ABEND-FIELD.9900-PARANAME, 1400-FILE-STATUS.1400-URM-STATUS.1400-URM-STATUS-OK ,
9900-ABEND-FIELD.9900-FILENAME , 9900-ABEND-FIELD.9900-STAT-CODE , 9900-ABEND-FIELD ,
1400-FILE-STATUS.1400-USOC-STATUS.1400-USOC-STATUS-OK )
    CALL 9000-READ-URM( 1000-URM-DETAIL.1000-URM-USOC.1000-URM-TRAILER , 100-URM-FILE.ELEMENT1.ELEMENT1,
100-URM-FILE.INDEXELEMENT1.INDEXELEMENT1 , 1000-URM-DETAIL.1000-URM-CLASS , 100-URM-FILE.ELEMENT2 ,
100-URM-FILE.INDEXELEMENT2 , 1000-URM-DETAIL.1000-URM-COS-DESCRIPTION , 100-URM-FILE.ELEMENT3 ,
100-URM-FILE.INDEXELEMENT3 , 1000-URM-DETAIL.1000-URM-USOC-DESCRIPTION , 100-URM-FILE.ELEMENT4 ,
100-URM-FILE.INDEXELEMENT4 , 1000-URM-DETAIL.1000-URM-CURRENT-PRERATE-IND , 100-URM-FILE.ELEMENT5 ,
100-URM-FILE.INDEXELEMENT5 , 1000-URM-DETAIL.1000-URM-RATES[1].1000-USOC-BY-RG ,
100-URM-FILE.ELEMENT6.ELEMENT1 , 100-URM-FILE.INDEXELEMENT6.INDEXELEMENT1,
1000-URM-DETAIL.1000-URM-RATES[2].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT2 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT2 , 1000-URM-DETAIL.1000-URM-RATES[3].1000-USOC-BY-RG ,
100-URM-FILE.ELEMENT6.ELEMENT3 , 100-URM-FILE.INDEXELEMENT6.INDEXELEMENT3 ,
1000-URM-DETAIL.1000-URM-RATES[4].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT4 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT4 , 1000-URM-DETAIL.1000-URM-RATES[5].1000-USOC-BY-RG ,
100-URM-FILE.ELEMENT6.ELEMENT5 , 100-URM-FILE.INDEXELEMENT6.INDEXELEMENT5 ,
1000-URM-DETAIL.1000-URM-RATES[6].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT6 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT6 , 1000-URM-DETAIL.1000-URM-RATES[7].1000-USOC-BY-RG ,
100-URM-FILE.ELEMENT6.ELEMENT7 , 100-URM-FILE.INDEXELEMENT6.INDEXELEMENT7 , 1000-URM-DETAIL.FILLER1 ,
100-URM-FILE.ELEMENT7 , 100-URM-FILE.INDEXELEMENT7, 1400-FILE-STATUS.1400-URM-STATUS.1400-URM-STATUS-OK ,
9900-ABEND-FIELD.9900-STAT-OPCODE ,  9900-ABEND-FIELD.9900-FILENAME , 9900-ABEND-FIELD.9900-PARANAME ,
9900-ABEND-FIELD.9900-STAT-CODE , 9900-ABEND-FIELD ,1000-URM-DETAIL.1000-URM-USOC.1000-URM-TRAILER ,
100-URM-FILE.ELEMENT1.ELEMENT1, 100-URM-FILE.INDEXELEMENT1.INDEXELEMENT1 ,
1000-URM-DETAIL.1000-URM-CLASS , 100-URM-FILE.ELEMENT2 , 100-URM-FILE.INDEXELEMENT2 ,
1000-URM-DETAIL.1000-URM-COS-DESCRIPTION , 100-URM-FILE.ELEMENT3 , 100-URM-FILE.INDEXELEMENT3 ,
1000-URM-DETAIL.1000-URM-USOC-DESCRIPTION , 100-URM-FILE.ELEMENT4 , 100-URM-FILE.INDEXELEMENT4 ,
1000-URM-DETAIL.1000-URM-CURRENT-PRERATE-IND , 100-URM-FILE.ELEMENT5 , 100-URM-FILE.INDEXELEMENT5 ,
1000-URM-DETAIL.1000-URM-RATES[1].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT1 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT1, 1000-URM-DETAIL.1000-URM-RATES[2].1000-USOC-BY-RG ,
100-URM-FILE.ELEMENT6.ELEMENT2 , 100-URM-FILE.INDEXELEMENT6.INDEXELEMENT2 ,
1000-URM-DETAIL.1000-URM-RATES[3].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT3 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT3 , 1000-URM-DETAIL.1000-URM-RATES[4].1000-USOC-BY-RG ,
100-URM-FILE.ELEMENT6.ELEMENT4 , 100-URM-FILE.INDEXELEMENT6.INDEXELEMENT4 ,
1000-URM-DETAIL.1000-URM-RATES[5].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT5 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT5 , 1000-URM-DETAIL.1000-URM-RATES[6].1000-USOC-BY-RG ,
100-URM-FILE.ELEMENT6.ELEMENT6 , 100-URM-FILE.INDEXELEMENT6.INDEXELEMENT6 ,
1000-URM-DETAIL.1000-URM-RATES[7].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT7 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT7 , 1000-URM-DETAIL.FILLER1 , 100-URM-FILE.ELEMENT7 ,
100-URM-FILE.INDEXELEMENT7, 1400-FILE-STATUS.1400-URM-STATUS.1400-URM-STATUS-OK ,
9900-ABEND-FIELD.9900-STAT-OPCODE ,  9900-ABEND-FIELD.9900-FILENAME , 9900-ABEND-FIELD.9900-PARANAME ,
9900-ABEND-FIELD.9900-STAT-CODE ,   )
  END.

 PROC 1100-OPEN-FILES( 100-URM-FILE , 200-USOC-FILE , 9900-ABEND-FIELD.9900-STAT-OPCODE ,
9900-ABEND-FIELD.9900-PARANAME, 1400-FILE-STATUS.1400-URM-STATUS.1400-URM-STATUS-OK ,
9900-ABEND-FIELD.9900-FILENAME , 9900-ABEND-FIELD.9900-STAT-CODE , 9900-ABEND-FIELD ,
1400-FILE-STATUS.1400-USOC-STATUS.1400-USOC-STATUS-OK  )
   BEGIN
    100-URM-FILE.ISOPEN := 'TRUE'
    100-URM-FILE.FILEMODE := 'INPUT'
    200-USOC-FILE.ISOPEN := 'TRUE'
    200-USOC-FILE.FILEMODE := 'OUTPUT'

   9900-ABEND-FIELD.9900-STAT-OPCODE := 'OPEN '
   9900-ABEND-FIELD.9900-PARANAME := '1100-INITILIZE'
    IF NOT (1400-FILE-STATUS.1400-URM-STATUS.1400-URM-STATUS-OK) THEN
        9900-ABEND-FIELD.9900-FILENAME : = 'URM '
        9900-ABEND-FIELD.9900-STAT-CODE := 1400-URM-STATUS
       CALL U076(9900-ABEND-FIELD)
```

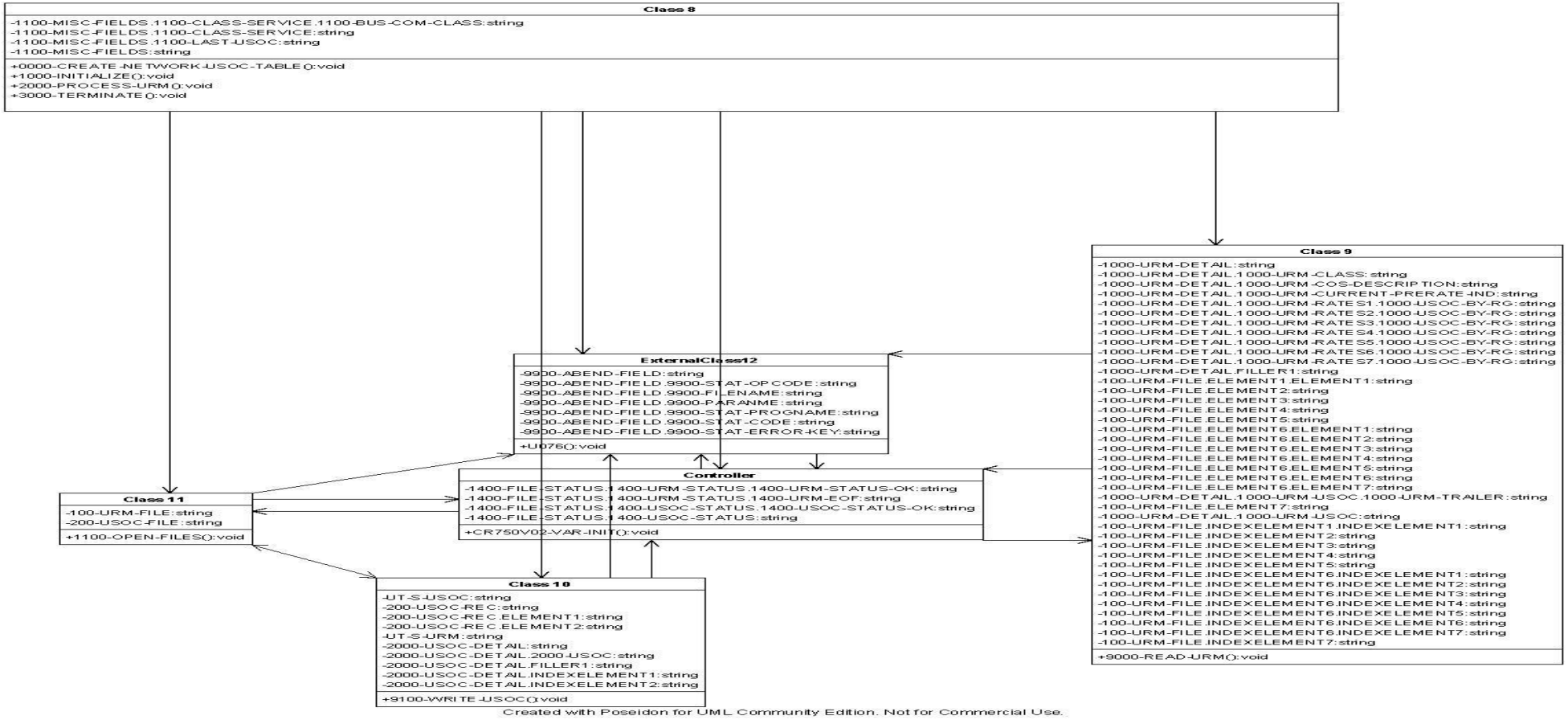
```
    FI
    IF  NOT (1400-FILE-STATUS.1400-USOC-STATUS.1400-USOC-STATUS-OK) THEN
       9900-ABEND-FIELD.9900-FILENAME := 'USOC'
       9900-ABEND-FIELD.9900-STAT-CODE := 1400-USOC-STATUS
       CALL U076(9900-ABEND-FIELD)
 END.

 PROC 2000-PROCESS-URM( 1100-MISC-FIELDs.1100-CLASS-SERVICE , 1000-URM-DETAIL.1000-URM-COS-DESCRIPTION ,
1000-URM-DETAIL.1000-URM-RATES[1].1000-USOC-BY-RG , 1000-URM-DETAIL.1000-URM-RATES[2].1000-USOC-BY-RG ,
1000-URM-DETAIL.1000-URM-RATES[1].1000-USOC-BY-RG , 1000-URM-DETAIL.1000-URM-USOC ,
1100-MISC-FIELDS.1100-LAST-USOC , 1100-MISC-FIELDS.1100-CLASS-SERVICE.1100-BUS-COM-CLASS ,
1100-MISC-FIELDS.1100-LAST-USOC , 1000-URM-DETAIL.1000-URM-USOC , 2000-USOC-DETAIL.2000-USOC ,
1100-MISC-FIELDS.1100-LAST-USOC , 1000-URM-DETAIL.1000-URM-USOC.1000-URM-TRAILER ,
100-URM-FILE.ELEMENT1.ELEMENT1, 100-URM-FILE.INDEXELEMENT1.INDEXELEMENT1 ,
1000-URM-DETAIL.1000-URM-CLASS , 100-URM-FILE.ELEMENT2 , 100-URM-FILE.INDEXELEMENT2 ,
1000-URM-DETAIL.1000-URM-COS-DESCRIPTION , 100-URM-FILE.ELEMENT3 , 100-URM-FILE.INDEXELEMENT3 ,
1000-URM-DETAIL.1000-URM-USOC-DESCRIPTION , 100-URM-FILE.ELEMENT4 , 100-URM-FILE.INDEXELEMENT4 ,
1000-URM-DETAIL.1000-URM-CURRENT-PRERATE-IND , 100-URM-FILE.ELEMENT5 , 100-URM-FILE.INDEXELEMENT5 ,
1000-URM-DETAIL.1000-URM-RATES[1].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT1 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT1, 1000-URM-DETAIL.1000-URM-RATES[2].1000-USOC-BY-RG ,
100-URM-FILE.ELEMENT6.ELEMENT2 , 100-URM-FILE.INDEXELEMENT6.INDEXELEMENT2 ,
1000-URM-DETAIL.1000-URM-RATES[3].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT3 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT3 , 1000-URM-DETAIL.1000-URM-RATES[4].1000-USOC-BY-RG ,
100-URM-FILE.ELEMENT6.ELEMENT4 , 100-URM-FILE.INDEXELEMENT6.INDEXELEMENT4 ,
1000-URM-DETAIL.1000-URM-RATES[5].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT5 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT5 , 1000-URM-DETAIL.1000-URM-RATES[6].1000-USOC-BY-RG ,
100-URM-FILE.ELEMENT6.ELEMENT6 , 100-URM-FILE.INDEXELEMENT6.INDEXELEMENT6 ,
1000-URM-DETAIL.1000-URM-RATES[7].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT7 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT7 , 1000-URM-DETAIL.FILLER1 , 100-URM-FILE.ELEMENT7 ,
100-URM-FILE.INDEXELEMENT7, 1400-FILE-STATUS.1400-URM-STATUS.1400-URM-STATUS-OK ,
9900-ABEND-FIELD.9900-STAT-OPCODE ,  9900-ABEND-FIELD.9900-FILENAME , 9900-ABEND-FIELD.9900-PARANAME ,
9900-ABEND-FIELD.9900-STAT-CODE , 9900-ABEND-FIELD , 200-USOC-REC.ELEMENT1 , 2000-USOC-DETAIL.2000-USOC ,
2000-USOC-DETAIL.INDEXELEMENT1,200-USOC-REC.ELEMENT2 , 2000-USOC-DETAIL.FILLER1 ,
2000-USOC-DETAIL.INDEXELEMENT2 , 1400-FILE-STATUS.1400-USOC-STATUS.1400-USOC-STATUS-OK ,
9900-ABEND-FIELD.9900-STAT-OPCODE , 9900-ABEND-FIELD.9900-PARANAME , 9900-ABEND-FIELD.9900-STAT-CODE ,
9900-ABEND-FIELD  )
  BEGIN
    1100-MISC-FIELDs.1100-CLASS-SERVICE := 1000-URM-DETAIL.1000-URM-COS-DESCRIPTION

    IF    1000-URM-DETAIL.1000-URM-RATES[1].1000-USOC-BY-RG <>
1000-URM-DETAIL.1000-URM-RATES[2].1000-USOC-BY-RG
     AND 1000-URM-DETAIL.1000-URM-RATES[1].1000-USOC-BY-RG >  0
     AND 1000-URM-DETAIL.1000-URM-USOC <>  1100-MISC-FIELDS.1100-LAST-USOC
     AND 1100-MISC-FIELDS.1100-CLASS-SERVICE.1100-BUS-COM-CLASS THEN
        1100-MISC-FIELDS.1100-LAST-USOC := 1000-URM-DETAIL.1000-URM-USOC
                                           2000-USOC-DETAIL.2000-USOC := 1100-MISC-FIELDS.1100-LAST-USOC
        CALL 9100-WRITE-USOC( 200-USOC-REC.ELEMENT1 , 2000-USOC-DETAIL.2000-USOC ,
2000-USOC-DETAIL.INDEXELEMENT1,200-USOC-REC.ELEMENT2 , 2000-USOC-DETAIL.FILLER1 ,
2000-USOC-DETAIL.INDEXELEMENT2 , 1400-FILE-STATUS.1400-USOC-STATUS.1400-USOC-STATUS-OK ,
9900-ABEND-FIELD.9900-STAT-OPCODE , 9900-ABEND-FIELD.9900-PARANAME , 9900-ABEND-FIELD.9900-STAT-CODE ,
9900-ABEND-FIELD )
    FI
    CALL 9000-READ-URM( 1000-URM-DETAIL.1000-URM-USOC.1000-URM-TRAILER , 100-URM-FILE.ELEMENT1.ELEMENT1,
100-URM-FILE.INDEXELEMENT1.INDEXELEMENT1 , 1000-URM-DETAIL.1000-URM-CLASS , 100-URM-FILE.ELEMENT2 ,
100-URM-FILE.INDEXELEMENT2 , 1000-URM-DETAIL.1000-URM-COS-DESCRIPTION , 100-URM-FILE.ELEMENT3 ,
100-URM-FILE.INDEXELEMENT3 , 1000-URM-DETAIL.1000-URM-USOC-DESCRIPTION , 100-URM-FILE.ELEMENT4 ,
100-URM-FILE.INDEXELEMENT4 , 1000-URM-DETAIL.1000-URM-CURRENT-PRERATE-IND , 100-URM-FILE.ELEMENT5 ,
100-URM-FILE.INDEXELEMENT5 , 1000-URM-DETAIL.1000-URM-RATES[1].1000-USOC-BY-RG ,
100-URM-FILE.ELEMENT6.ELEMENT1 , 100-URM-FILE.INDEXELEMENT6.INDEXELEMENT1,
1000-URM-DETAIL.1000-URM-RATES[2].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT2 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT2 , 1000-URM-DETAIL.1000-URM-RATES[3].1000-USOC-BY-RG ,
100-URM-FILE.ELEMENT6.ELEMENT3 , 100-URM-FILE.INDEXELEMENT6.INDEXELEMENT3 ,
1000-URM-DETAIL.1000-URM-RATES[4].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT4 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT4 , 1000-URM-DETAIL.1000-URM-RATES[5].1000-USOC-BY-RG ,
100-URM-FILE.ELEMENT6.ELEMENT5 , 100-URM-FILE.INDEXELEMENT6.INDEXELEMENT5 ,
1000-URM-DETAIL.1000-URM-RATES[6].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT6 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT6 , 1000-URM-DETAIL.1000-URM-RATES[7].1000-USOC-BY-RG ,
100-URM-FILE.ELEMENT6.ELEMENT7 , 100-URM-FILE.INDEXELEMENT6.INDEXELEMENT7 , 1000-URM-DETAIL.FILLER1 ,
100-URM-FILE.ELEMENT7 , 100-URM-FILE.INDEXELEMENT7, 1400-FILE-STATUS.1400-URM-STATUS.1400-URM-STATUS-OK ,
9900-ABEND-FIELD.9900-STAT-OPCODE ,  9900-ABEND-FIELD.9900-FILENAME , 9900-ABEND-FIELD.9900-PARANAME ,
9900-ABEND-FIELD.9900-STAT-CODE , 9900-ABEND-FIELD )
  END.


 PROC 3000-TERMINATE( 100-URM-FILE , 200-USOC-FILE , 9900-ABEND-FIELDS.9900-STAT-OPCODE ,
9900-ABEND-FIELDS.9900-PARANAME , 1400-FILE-STATUS.1400-URM-STATUS.1400-URM-STATUS-OK ,
9900-ABEND-FIELDS.9900-FILENAME , 9900-ABEND-FIELDS.9900-STAT-CODE , 9900-ABEND-FIELD ,
1400-FILE-STATUS.1400-USOC-STATUS.1400-USOC-STATUS-OK    )
  BEGIN
```

```
    100-URM-FILE.ISOPEN := 'FALSE'
    200-USOC-FILE.ISOPEN := 'FALSE'

    9900-ABEND-FIELDS.9900-STAT-OPCODE := 'CLOSE'
    9900-ABEND-FIELDS.9900-PARANAME := '3000-TERMINATE'

   IF  NOT (1400-FILE-STATUS.1400-URM-STATUS.1400-URM-STATUS-OK)  THEN
       9900-ABEND-FIELDS.9900-FILENAME := 'URM '
       9900-ABEND-FIELDS.9900-STAT-CODE := 1400-FILE-STATUS.1400-URM-STATUS
       CALL U076(9900-ABEND-FIELD)
   FI

   IF  NOT (1400-FILE-STATUS.1400-USOC-STATUS.1400-USOC-STATUS-OK)   THEN
       9900-ABEND-FIELDS.9900-FILENAME := 'USOC'
       9900-ABEND-FIELDS.9900-STAT-CODE := 1400-FILE-STATUS.1400-USOC-STATUS
       CALL U076(9900-ABEND-FIELD)
   FI
 END.

  PROC 9000-READ-URM( 1000-URM-DETAIL.1000-URM-USOC.1000-URM-TRAILER , 100-URM-FILE.ELEMENT1.ELEMENT1,
100-URM-FILE.INDEXELEMENT1.INDEXELEMENT1 , 1000-URM-DETAIL.1000-URM-CLASS , 100-URM-FILE.ELEMENT2 ,
100-URM-FILE.INDEXELEMENT2 , 1000-URM-DETAIL.1000-URM-COS-DESCRIPTION , 100-URM-FILE.ELEMENT3 ,
100-URM-FILE.INDEXELEMENT3 , 1000-URM-DETAIL.1000-URM-USOC-DESCRIPTION , 100-URM-FILE.ELEMENT4 ,
100-URM-FILE.INDEXELEMENT4 , 1000-URM-DETAIL.1000-URM-CURRENT-PRERATE-IND , 100-URM-FILE.ELEMENT5 ,
100-URM-FILE.INDEXELEMENT5 , 1000-URM-DETAIL.1000-URM-RATES[1].1000-USOC-BY-RG ,
100-URM-FILE.ELEMENT6.ELEMENT1 , 100-URM-FILE.INDEXELEMENT6.INDEXELEMENT1,
1000-URM-DETAIL.1000-URM-RATES[2].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT2 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT2 , 1000-URM-DETAIL.1000-URM-RATES[3].1000-USOC-BY-RG ,
100-URM-FILE.ELEMENT6.ELEMENT3 , 100-URM-FILE.INDEXELEMENT6.INDEXELEMENT3 ,
1000-URM-DETAIL.1000-URM-RATES[4].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT4 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT4 , 1000-URM-DETAIL.1000-URM-RATES[5].1000-USOC-BY-RG ,
100-URM-FILE.ELEMENT6.ELEMENT5 , 100-URM-FILE.INDEXELEMENT6.INDEXELEMENT5 ,
1000-URM-DETAIL.1000-URM-RATES[6].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT6 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT6 , 1000-URM-DETAIL.1000-URM-RATES[7].1000-USOC-BY-RG ,
100-URM-FILE.ELEMENT6.ELEMENT7 , 100-URM-FILE.INDEXELEMENT6.INDEXELEMENT7 , 1000-URM-DETAIL.FILLER1 ,
100-URM-FILE.ELEMENT7 , 100-URM-FILE.INDEXELEMENT7, 1400-FILE-STATUS.1400-URM-STATUS.1400-URM-STATUS-OK ,
9900-ABEND-FIELD.9900-STAT-OPCODE ,  9900-ABEND-FIELD.9900-FILENAME , 9900-ABEND-FIELD.9900-PARANAME ,
9900-ABEND-FIELD.9900-STAT-CODE , 9900-ABEND-FIELD )
   BEGIN
     FETCH 1000-URM-DETAIL.1000-URM-USOC.1000-URM-TRAILER , 100-URM-FILE.ELEMENT1.ELEMENT1,
100-URM-FILE.INDEXELEMENT1.INDEXELEMENT1
     FETCH 1000-URM-DETAIL.1000-URM-CLASS , 100-URM-FILE.ELEMENT2 , 100-URM-FILE.INDEXELEMENT2
     FETCH 1000-URM-DETAIL.1000-URM-COS-DESCRIPTION , 100-URM-FILE.ELEMENT3 , 100-URM-FILE.INDEXELEMENT3
     FETCH 1000-URM-DETAIL.1000-URM-USOC-DESCRIPTION , 100-URM-FILE.ELEMENT4 , 100-URM-FILE.INDEXELEMENT4
     FETCH 1000-URM-DETAIL.1000-URM-CURRENT-PRERATE-IND , 100-URM-FILE.ELEMENT5 ,
100-URM-FILE.INDEXELEMENT5
     FETCH 1000-URM-DETAIL.1000-URM-RATES[1].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT1 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT1
     FETCH 1000-URM-DETAIL.1000-URM-RATES[2].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT2 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT2
     FETCH 1000-URM-DETAIL.1000-URM-RATES[3].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT3 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT3
     FETCH 1000-URM-DETAIL.1000-URM-RATES[4].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT4 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT4
     FETCH 1000-URM-DETAIL.1000-URM-RATES[5].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT5 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT5
     FETCH 1000-URM-DETAIL.1000-URM-RATES[6].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT6 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT6
     FETCH 1000-URM-DETAIL.1000-URM-RATES[7].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT7 ,
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT7
     FETCH 1000-URM-DETAIL.FILLER1 , 100-URM-FILE.ELEMENT7 , 100-URM-FILE.INDEXELEMENT7


     /*  READ 100-URM-FILE INTO 1000-URM-DETAIL. */

     IF  NOT 1400-FILE-STATUS.1400-URM-STATUS.1400-URM-STATUS-OK THEN
        9900-ABEND-FIELD.9900-STAT-OPCODE := 'READ '
        9900-ABEND-FIELD.9900-FILENAME := 'URM  '
        9900-ABEND-FIELD.9900-PARANAME := '9000-READ-URM'
        9900-ABEND-FIELD.9900-STAT-CODE := 1400-FILE-STATUS.1400-URM-STATUS
        CALL  U076(9900-ABEND-FIELD)
      FI
   END.

 PROC 9100-WRITE-USOC(200-USOC-REC.ELEMENT1 , 2000-USOC-DETAIL.2000-USOC ,
2000-USOC-DETAIL.INDEXELEMENT1,200-USOC-REC.ELEMENT2 , 2000-USOC-DETAIL.FILLER1 ,
2000-USOC-DETAIL.INDEXELEMENT2 , 1400-FILE-STATUS.1400-USOC-STATUS.1400-USOC-STATUS-OK ,
9900-ABEND-FIELD.9900-STAT-OPCODE , 9900-ABEND-FIELD.9900-PARANAME , 9900-ABEND-FIELD.9900-STAT-CODE ,
9900-ABEND-FIELD )
```

```
 BEGIN

    PUT  200-USOC-REC.ELEMENT1, 2000-USOC-DETAIL.2000-USOC , 2000-USOC-DETAIL.INDEXELEMENT1
    PUT  200-USOC-REC.ELEMENT2, 2000-USOC-DETAIL.FILLER1 ,  2000-USOC-DETAIL.INDEXELEMENT2

   /* WRITE 200-USOC-REC FROM 2000-USOC-DETAIL. */


   IF  NOT 1400-FILE-STATUS.1400-USOC-STATUS.1400-USOC-STATUS-OK THEN
      9900-ABEND-FIELD.9900-STAT-OPCODE := 'READ'
      9900-ABEND-FIELD.9900-FILENAME := 'USOC'

  FI

END.
```

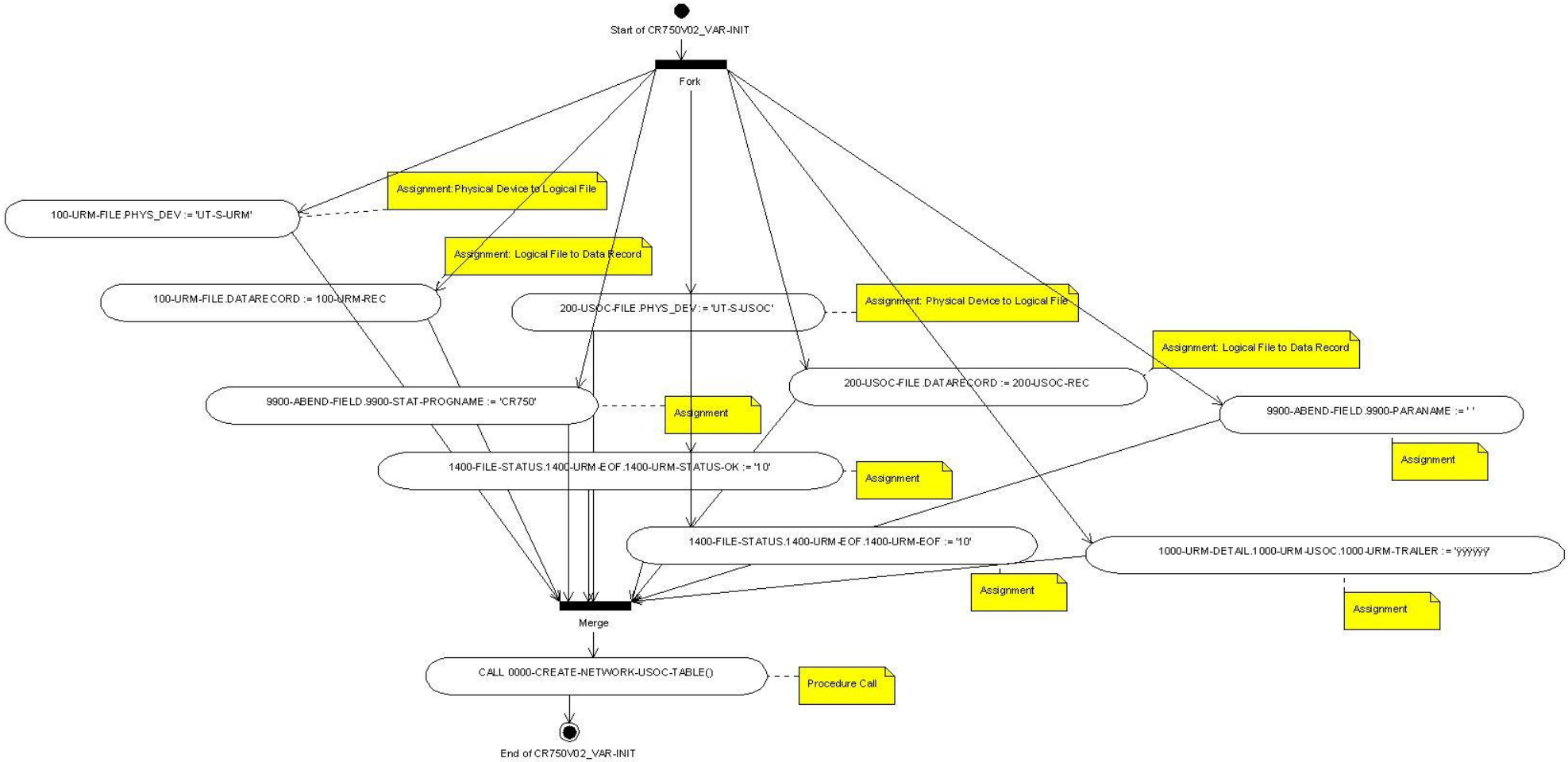
Appendix C: Extracted UML Diagrams from a Selected Sample(CR750V02)

Class Diagram

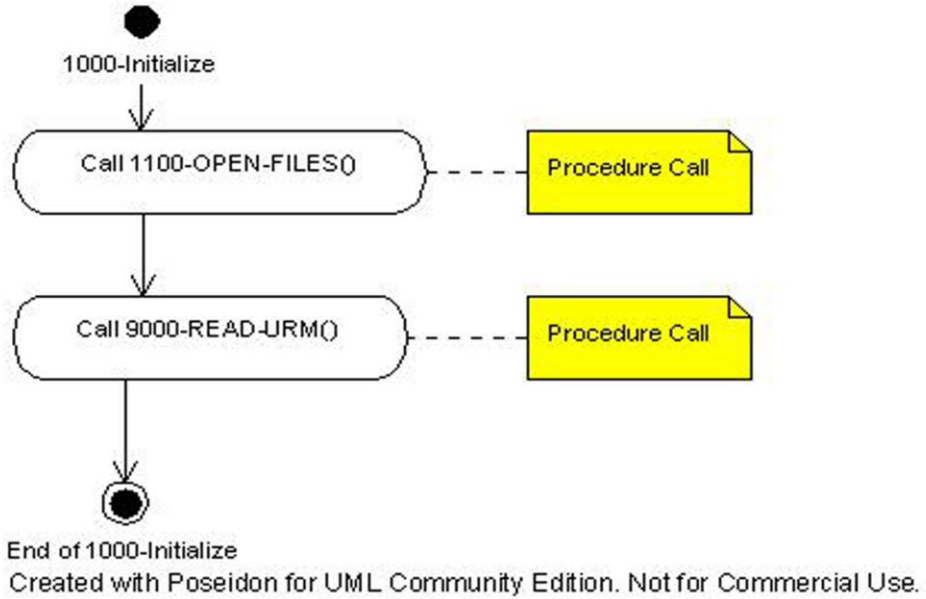


Activity Diagrams of CR750V02

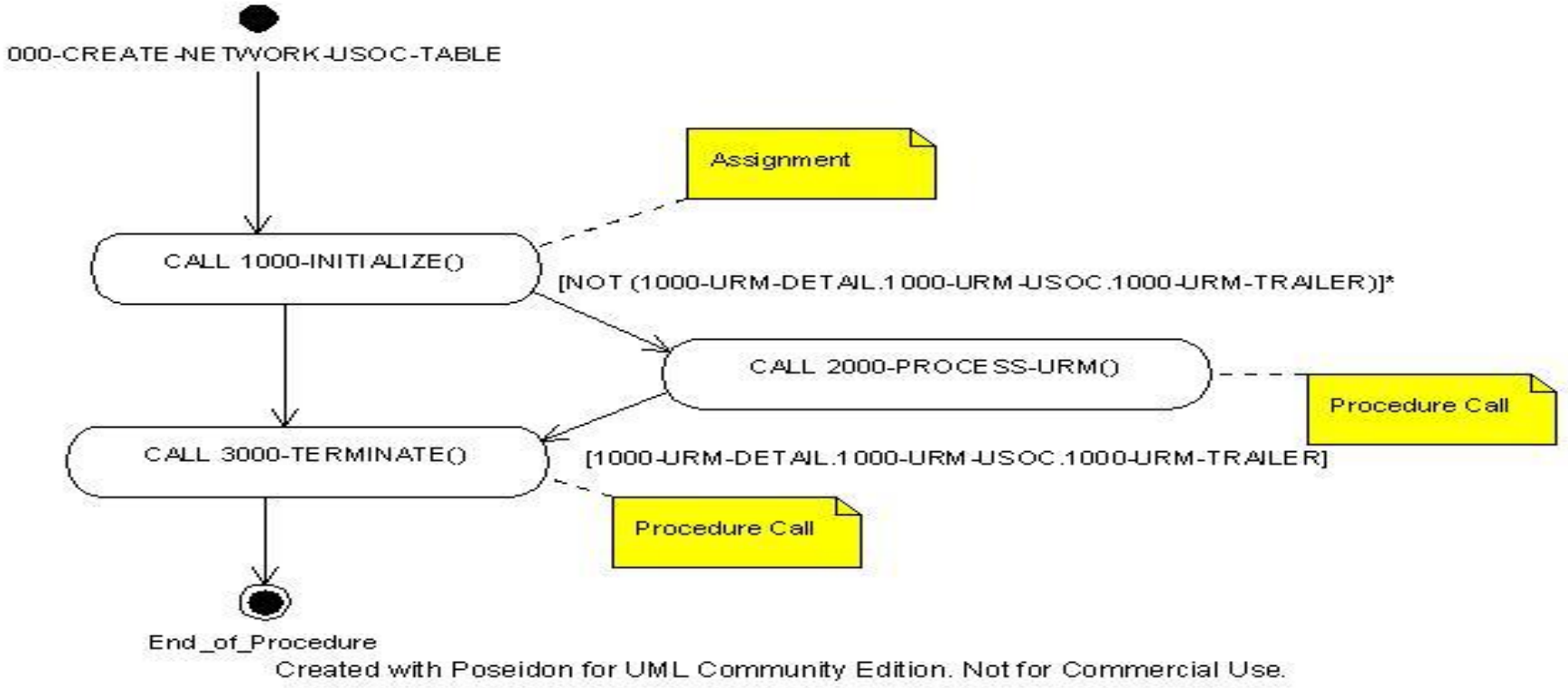
CR750V02_VAR-INIT



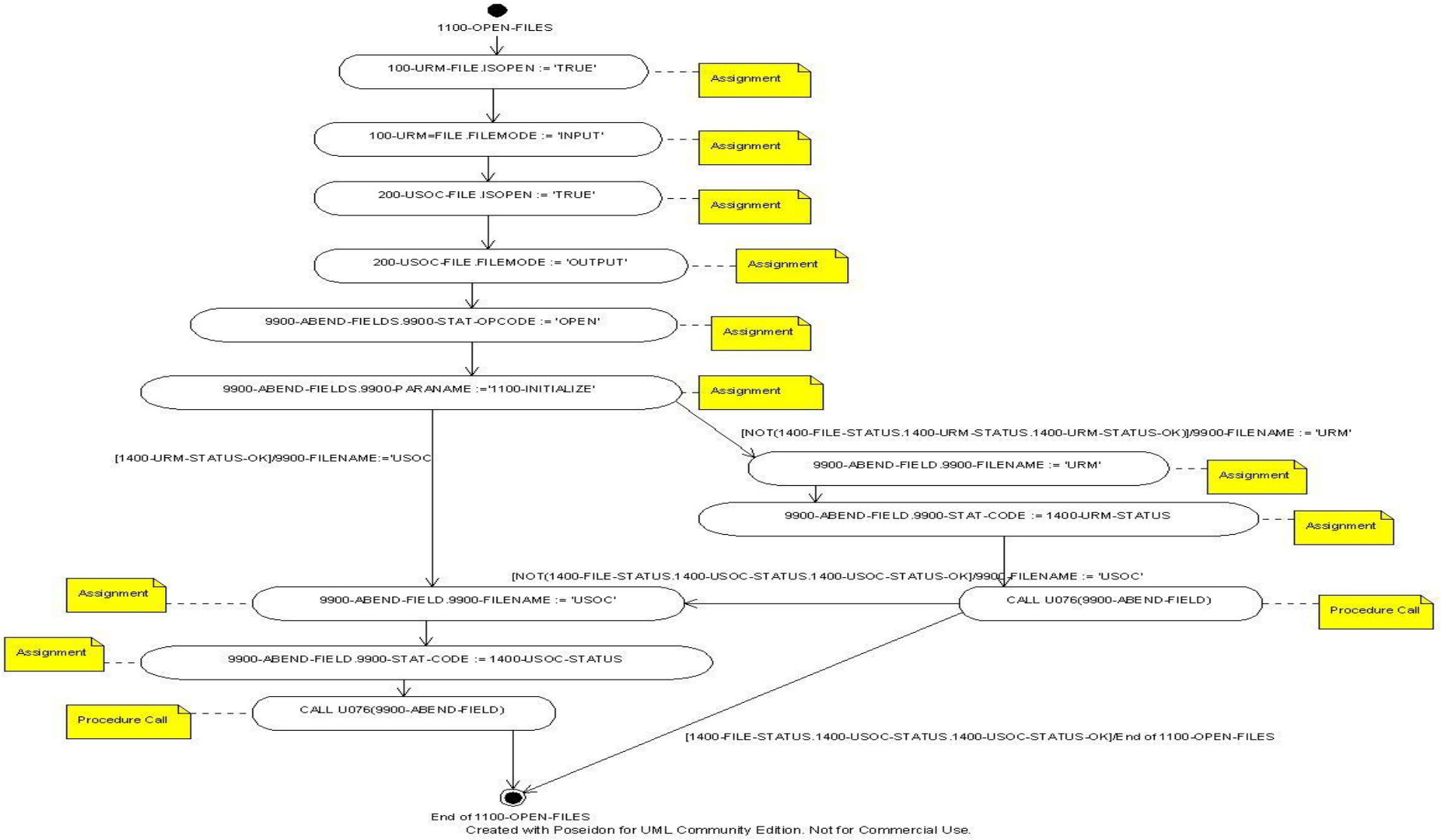
1000-INITIALIZE



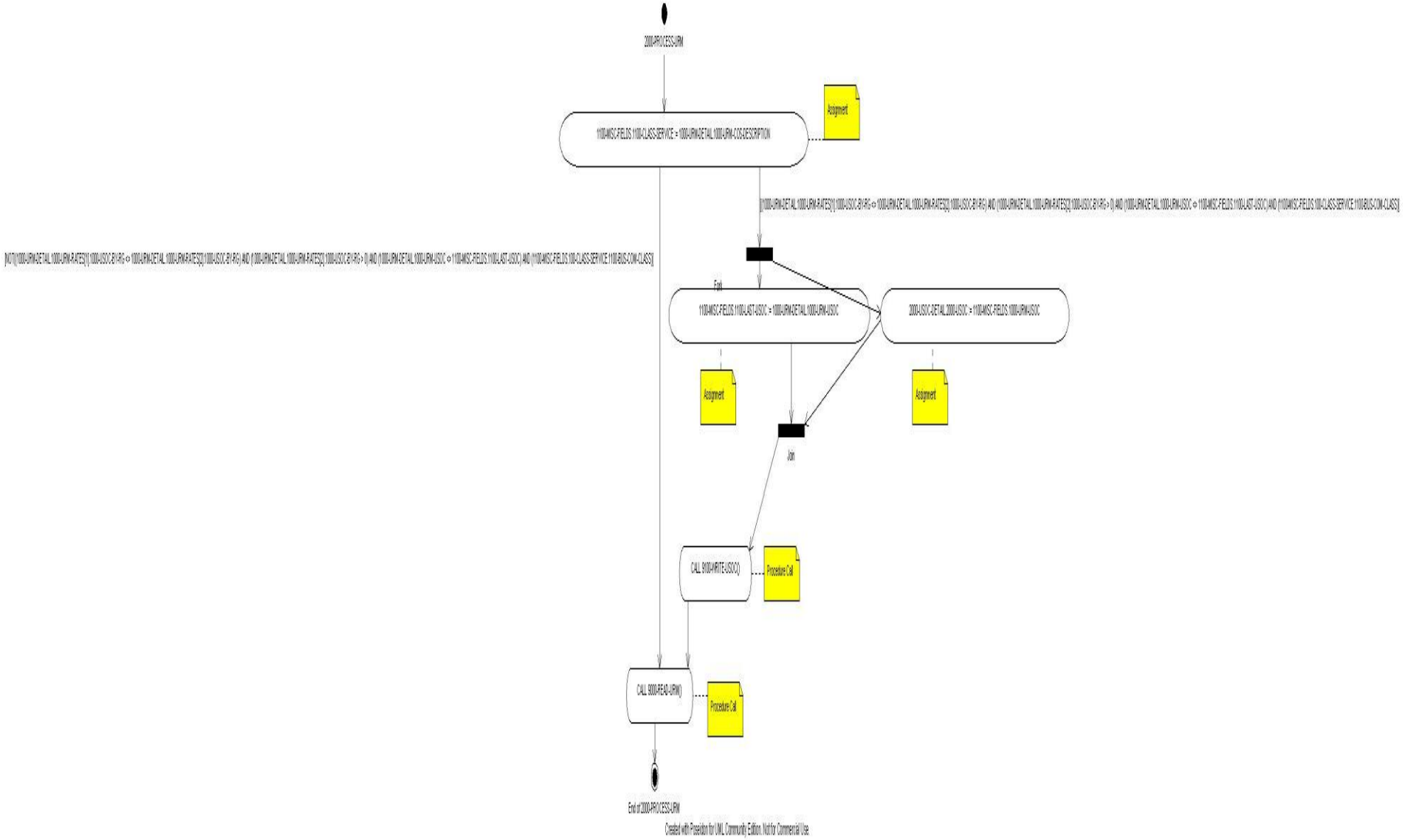
000-CREATE-NETWORK-USOC-TABLE



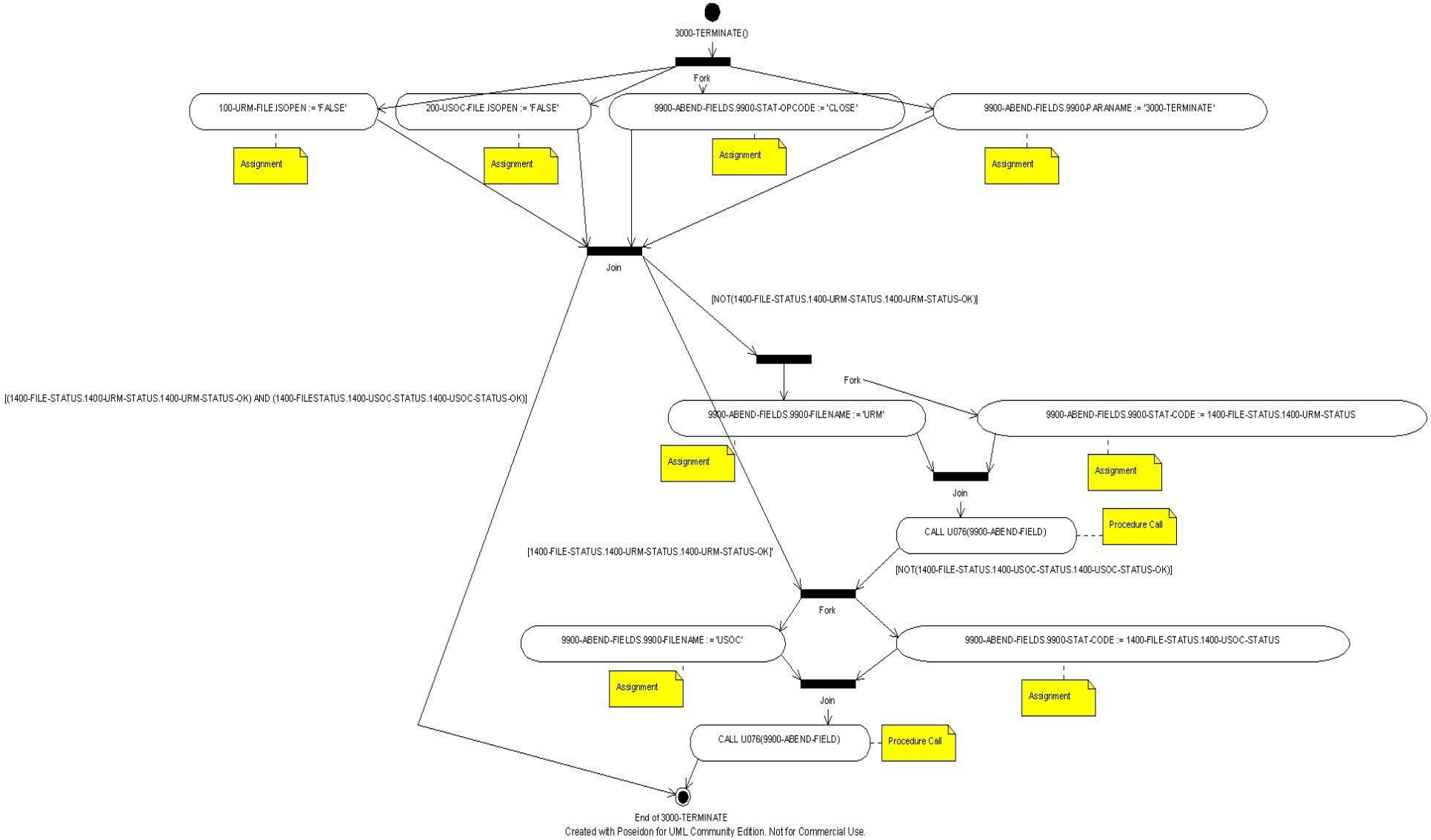
1100-OPEN-FILES



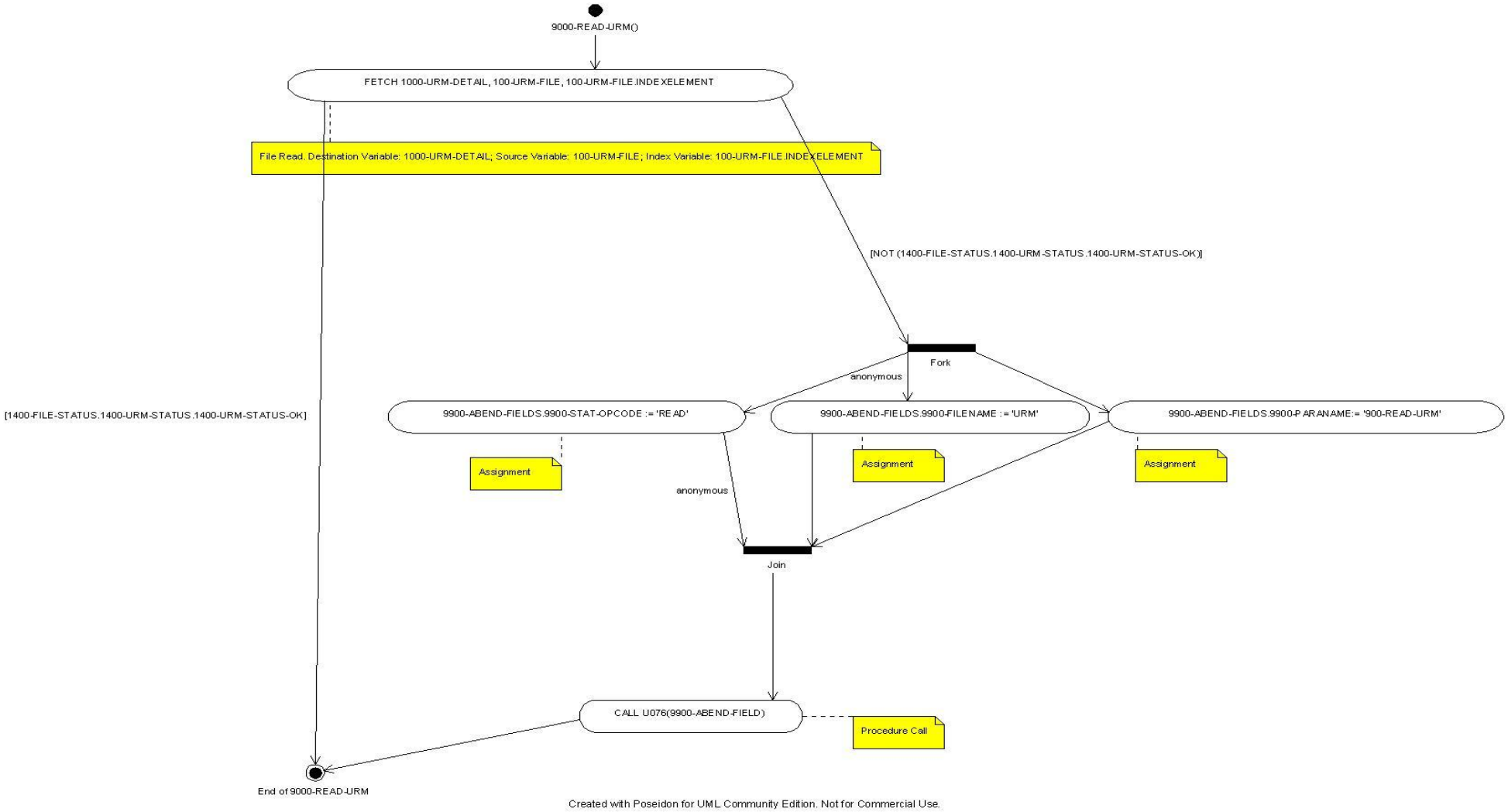
2000-PROCESS-URM



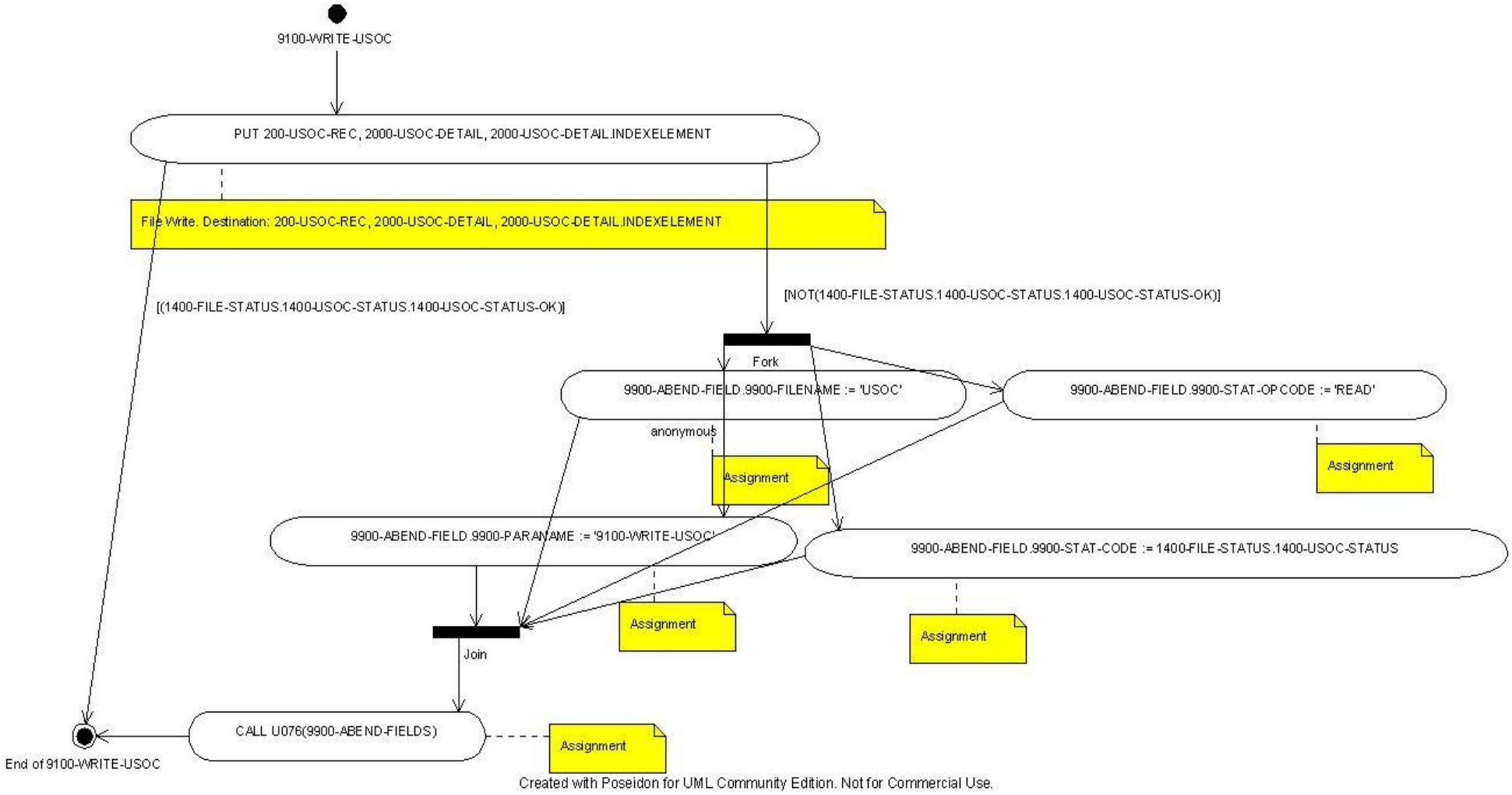
3000-TERMINATE



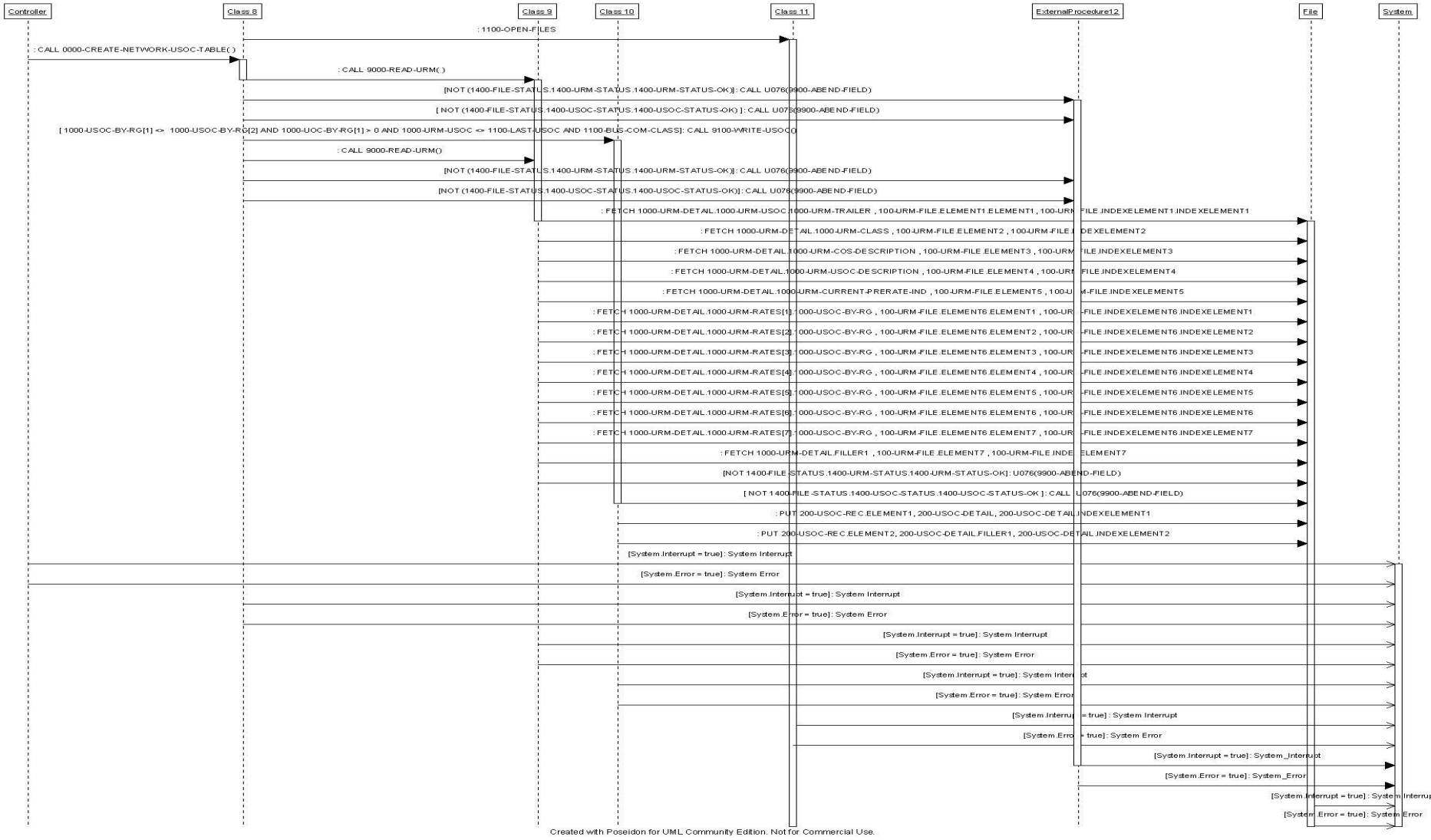
9000-READ-URM



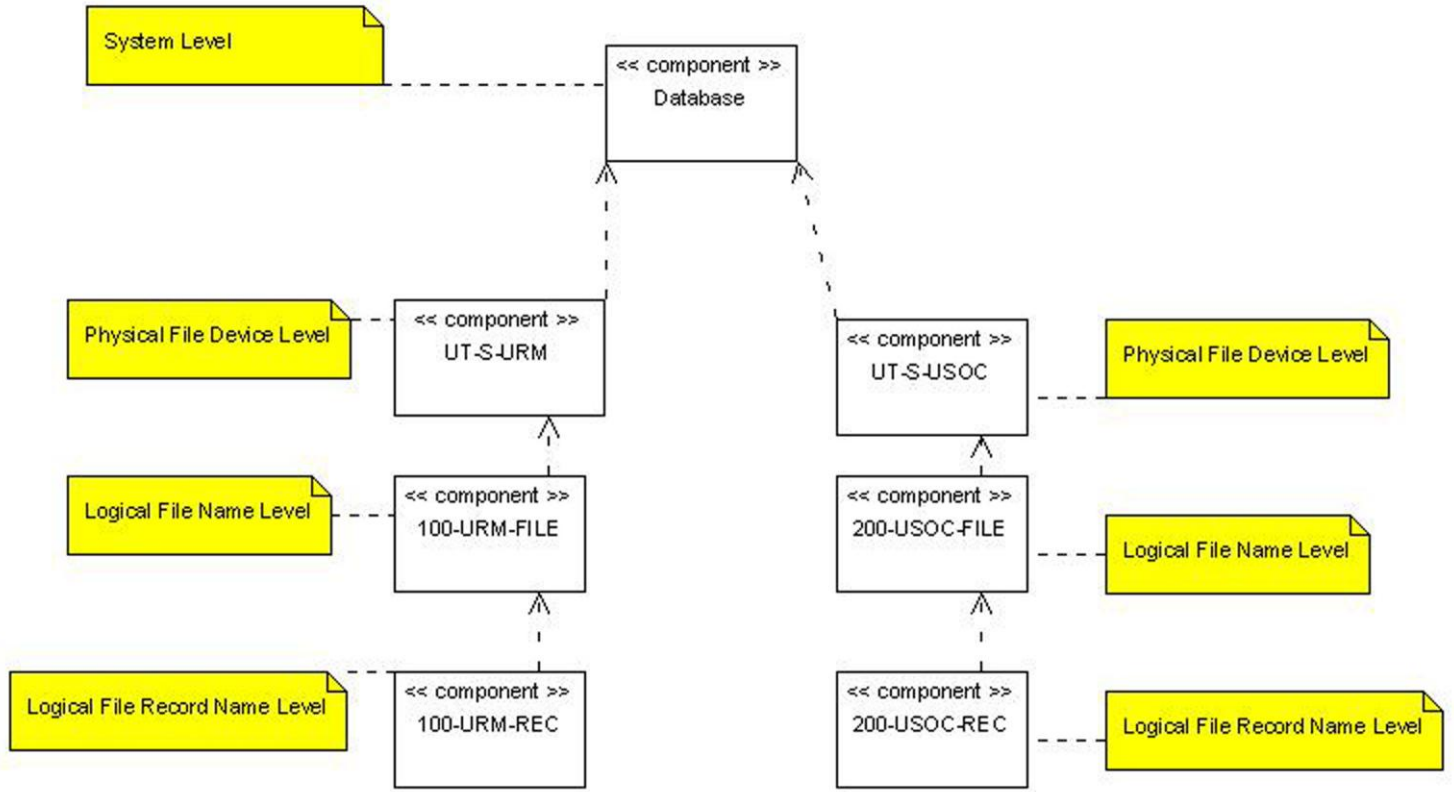
9100-WRITE-USOC



Sequence Diagram of CR750V02



Component Diagram of CR750V02



Appendix D: Conversion Rules and Sample from WSL to C++

D.1 Introduction and the Thesis Approach

One feature of the TAGDUR tool, not yet fully developed, is the ability to generate C++ code from its intermediate WSL representation of a system. This C++ code generation involves many issues such as the weak data typing of COBOL versus the strong data typing of C++. One of the benefits of a full-fledged code generator within a reengineering tool is that it saves a tremendous amount of development time and cost while avoiding many of the mistakes that a manual re-coding might involve, particularly in regards to dealing with legacy COBOL systems' numerous similar-spelled variable names.

Many reengineering tools have a round-trip re-engineering feature. However, many tools base their code regeneration on producing class skeletons, with procedure headers and attribute declarations, from the UML class diagrams (Greefhorst, 1998). One tool, Describe from Embercardo Technologies, has the ability to generate code to represent procedure calls from its tool's generated sequence diagram (Embercardo, 2004). However, these tools require that developers manually code the remaining procedural or class code. Univan and George (Univan, 2000) outline methods to develop a C++ translator for the RAISE specification language. RAISE, although a wide spectrum language, is a typed language, unlike the WSL used in this thesis. Typing of variables is most useful in the specifications for forward engineering, before a particular implementation language has been chosen, and not in reverse engineering when the particular data type of a variable needs to be deciphered from its source code and usage. An example, a Boolean variable can be represented, in some systems, as an integer rather than a Boolean type.

After the translation of the selected legacy system from COBOL to WSL has been completed, the WSL representation is translated to C++. This WSL-C++ translation allows for source programming language independence but it also requires the original source programming data typing, in COBOL in this instance, to be converted into C++ data types. C++ code generation from a WSL system representation also involves implementation-dependent issues such as dynamic memory allocation, file handling, file granularity, and independent task allocation. These issues are addressed in this section.

D.2 Dynamic Memory Allocation

There are important differences between how COBOL, WSL, and C++ deals with memory allocation for its variables. COBOL utilises static memory allocation for its variables. In COBOL, memory automatically is allocated for variables, regardless of their type, as soon as the variables are declared. The memory allocated to variables is freed automatically when variables go out of scope which, because of the global scope of all variables in COBOL, occurs when the program ends. WSL, because it is implementation detail independent, does not deal with memory allocation of its variables. Once variables are declared in WSL, there is no need to allocate memory for them. Nor is there any need to deallocate this memory at a later point in the program. Memory allocation is an implementation detail specific issue while WSL is implementation detail independent. C++, however, is implementation detail dependent. C++ automatically allocates memory for variables that are not of type class, string, or struct and then deallocates this memory whenever these variables go out of scope. In C++, variables of type class, string, or struct must either be initialised or have memory dynamically allocated to them. This allocated memory is deallocated when these variables go out of scope.

Because the C++ code generation of our thesis tool does not have sophisticated garbage collection that can detect when a dynamically allocated variable is first needed and when this variable is no longer needed, dynamically allocated variables are allocated memory, via the malloc function call, in the initialization method of the class whose attributes are these variables. The memory allocated to these variables is then freed, via a free function call, in the termination method of this same class. Strings in C++ are declared as arrays of type char rather than as a pointer to a char. This pointer would then be allocated the memory needed to hold a string of maximum size using the C++ malloc function.

D.3 File Handling

COBOL, like many programming languages, assigns logical file names to a physical file. Each file record is assigned WSL-specific record fields that contain implementation-specific details, such as the file's file mode, the physical file name, and index key.

COBOL opens a file for a specific mode of file operation, whether this mode is for input, output, or append. Because WSL is implementation detail independent, WSL is not concerned with the file mode of the associated file but simply records it in the WSL-specific record fields associated with this file record. However, C++, like COBOL, is concerned with the file mode of any file. Consequently, the file mode of each file is retrieved from WSL and is then incorporated in the file open statement in C++ for each file.

D.4 File Granularity

Because WSL represents file I/O as an assignment between a variable and a physical shunt, the atomicity of file I/O is not predetermined. Consequently, this atomicity may be as fine as an individual record field. Because the granularity of a file operation is dependent on the atomicity of file I/O, atomicity at the individual record field level means that the granularity of a task is defined, and the independence of tasks is evaluated, at the individual file record field level.

Because file operations typically occur at the record level and because individual record fields of the same record typically have no shared data or control dependencies, all individual record fields of the same record in a file operation could be executed in parallel in a theoretical computer application. However, in C++, attempts by several processes to access the same file stream to update in parallel several individual record fields may cause file contention issues. Consequently, in C++,

file operations occur at the record level rather than the individual record field level in order to avoid these file contention issues.

D.5 Data Typing

Each variable, that are non-structure in nature, needs to have their data type declared in C++. When the legacy COBOL source code is parsed, the data type, declared as in the PIC clause, is kept in the database along with the variable name. During the code generation phase, this data type is looked up whenever the associated variable name is declared. The data type, originally in COBOL, is converted to its equivalent C++ data type.

The conversion from COBOL to C++ data type follows these set of rules:

1) if the COBOL PIC's clause contains a numeric mantissa, in the form of a "V" special character, the equivalent C++ data type would either be a *float* or *double*. An example, PIC 9999V99 in COBOL converts to a float in C++.

2) If the COBOL PIC's clause does not contain a mantissa but is numeric, in the form of a "9" special character, the equivalent C++ data type would be of type *int*. An example, PIC 9999 in COBOL converts to an int in C++.

3) If the COBOL PIC's clause is alphanumeric, denoted by the "X" special character, the equivalent C++ data type would be of type char or string of the same size as declared in the COBOL PIC clause. The size of the string as declared in the PIC clause is either the number of "X" special characters in the PIC clause or the number declared within the parentheses contained in the PIC clause. An example, PIC X(10) in COBOL would denote a string of maximum size 10 and is represented as char[10] in C++.

A particular problem emerges with date variables. In COBOL, date variables are declared of type string. In C++, date variables can be declared exclusively as type datetime. Internally, C++ represents dates/datetimes as numeric values rather than an alphanumeric value. However, without the use of extensive usage graphs, it is difficult to ascertain whether a date variable, declared as a string in COBOL and consequently translated as a string variable in C++, is really a date variable. Consequently, it was decided to keep the date variables in C++ as string with implicit date type conversion whenever this variable receives a value of type date, whether it is from a date system call or through an assignment.

One disadvantage in the conversion from COBOL to C++ data types is that this conversion loses the great precision that COBOL has with its numeric data types. COBOL allows you to precisely specify the size of the base and mantissa of a numeric data type variable. An example, a COBOL variable of PIC 9999V99 has a base of size four and a mantissa of size two. The equivalent C++ data type of this COBOL variable is of type float, which has a maximum size of 3.40282e+38 (Alvin, 2004). The C++ data type equivalent clearly lacks the precision in specifying the range of maximum values that a particular variable may hold.

This precise specification of a range of possible values that a variable may contain in COBOL presents a problem when converting this COBOL program to its C++ equivalent. An example, in a COBOL program, a variable, SIN, has a declared base of size nine with no mantissa. When a user enters a person's SIN (Social Insurance Number) from the keyboard, this entered value is then assigned to the SIN variable. The COBOL program then may check if a data overflow occurs when the entered value is assigned to the SIN variable. If a data overflow error occurs, the COBOL program may then display an error message of "Invalid SIN number". Otherwise, if there is no data overflow error, normal processing occurs. When this COBOL program is converted to its C++ equivalent, this data overflow check, which is really a data range check, must be converted to its C++ equivalent. In the previous example, the data overflow check is converted to a test to determine if the user-entered value is written within the permittable range of values (greater than zero but less than 1 000 000 000.

D.6 Exception Handling

The original COBOL source program had no provisions for error handling. When this program was converted to WSL, provision for system interrupts and error handling was made by enclosing the procedure code in a control construct which tested if a system interrupt or error had occurred. If a system interrupt or error occurred, the appropriate system interrupt or error handling function will be called. Enclosing each WSL programming statement by both an interrupt/error test and handler is an extension of Gerlich's rule (Gerlich, 2000) that an exception handler must be specified for every state in a system's statechart in order to handle commands or behaviour that is not specified in the transitions of that state.

Because C++, unlike some programming languages such as assembly language, does not require its system interrupts and interrupt handlers to be programmed explicitly, these system interrupts are not specified explicitly in the C++ program. In C++, all procedure code is given a default exception handler by enclosing all the procedure code within a try...catch block. This catch sub-block can also hold system exception handlers. A general system exception handler is assigned to each C++ procedure by default because the original COBOL source code had no system exception handlers.

System calls in WSL are translated to their equivalent in C++ when applicable. An example, the WSL system call in the format, <Destination_Variable> = SystemDate(), is modified to its C++ equivalent in the format <Destination_Variable> = Date.

D.7 General WSL translations to their C++ equivalents

Begin and do keywords are replaced by "{". End, fi, or od keywords are replaced by "}".

WSL comments are prefixed by "//" in order to convert these comments into C++ comments.

Assign statements in WSL, in the format of <Dest_Variable> := <Operand_Variable1> <Operator> <Operand_Variable2>, remain unchanged except for a ";" end of statement delimiter is affixed at the end of statement and the substitution of the C++ assignment operator, "=", for its WSL equivalent operator, ":=".

Control constructs, such as if or while statements, remain little changed from their WSL statements. A While statement's condition clause in WSL is enclosed by parenthesises. Similarly, an If statement's condition clause in WSL is enclosed by parenthesises. In any condition clause, the WSL operator, "=", is replaced by its C++ equivalent operator,"= ="

D.8 Example of WSL to C++ Conversion

A short example (132 WSL lines translated into 101 C++ lines of code) is given of part of a simple banking transaction system that updates and displays a customer's bank account.. This example demonstrates that WSL can be translated into C++ and that the enclosing system interrupt/error check constructs in WSL can be replaced by C++ simpler try-catch system exception handling blocks.

D.8.1 WSL Implementation:

```
Class ObjectA
 Begin
Var Struct FileRecord
  Begin
    Var Account1
    Var Account2
    Var Account3
 End
Var CurDate
Var Struct FileRecord_Index
  Begin
    Var Element1
    Var Element2
    Var Element3
  End
Var FileRecord_Index_Count
Proc Init
 Begin
  If (System.Interrupt_Status <> 'Interrupt') and (System.Error_Status <> 'Error') Then
   CurDate := SystemDate()
 Elif (System.Interrupt_Status = 'Interrupt') Then
  Call System_Interrupt()
 Else
  Call System_Error()
 Fi
 If (System.Interrupt_Status <> 'Interrupt') and (System.Error_Status <> 'Error') Then
   FileRecord.Phys_Dev := 'Acct.dat'
Elif (System.Interrupt_Status = 'Interrupt') Then
  Call System_Interrupt()
 Else
   Call System_Error()
 Fi
If (System.Interrupt_Status <> 'Interrupt') and (System.Error_Status <> 'Error') Then
   FileRecord.File_Mode := 'Input'
Elif (System.Interrupt _Status = 'Interrupt') Then
  Call System_Interrupt()
 Else
   Call System_Error()
 Fi
If (System.Interrupt_Status <> 'Interrupt') and (System.Error_Status <> 'Error') Then
   FileRecord_Index_Count := 1
Elif (System.Interrupt_Status = 'Interrupt') Then
  Call System_Interrupt()
Elif (System.Error_Status = 'Error') Then
   Call System_Error()
Fi
End
Proc ReadRec(Account1, Account2, Account3)
If (System.Interrupt_Status <> 'Interrupt) And (System.Error_Status <> 'Error) Then
 Begin
   If (System.Interrupt_Status <> 'Interrupt') and (System.Error_Status <> 'Error') Then
       Fetch Account1, FileRecord.Account1, FileRecord_Index.Element1
   Elif (System.Interrupt_Status = 'Interrupt') Then
     Call System_Interrupt()
   Elif (System.Error_Status = 'Error') Then
    Call System_Error()
   Fi
    If (System.Interrupt_Status <> 'Interrupt') and (System.Error_Status <> 'Error') Then
```

```
  Fetch Account2, FileRecord.Account2, FileRecord_Index.Element2
 Elif (System.Interrupt_Status = 'Interrupt') Then
   Call System_Interrupt()
 Elif (System.Error_Status = 'Error') Then
  Call System_Error()
 Fi
  If (System.Interrupt_Status <> 'Interrupt') and (System.Error_Status <> 'Error') Then
  Fetch Account3, FileRecord.Account3, FileRecord_Index.Element3
 Elif (System.Interrupt_Status = 'Interrupt') Then
   Call System_Interrupt()
 Elif (System.Error_Status = 'Error') Then
  Call System_Error()
 Fi
If (System.Interrupt_Status <> 'Interrupt') and (System.Error_Status <> 'Error') Then
  FileRecord_Index_Count := FileRecord_Index_Count + 1
Elif (System.Interrupt_Status = 'Interrupt') Then
   Call System_Interrupt()
Elif (System.Error_Status = 'Error') Then
  Call System_Error()
Fi
End
 Elif (System.Interrupt_Status = 'Interrupt) Then
  Call System_Interrupt()
 Elif (System.Error_Status = 'Error) Then
  Call System_Error()
 Fi
End

Class ObjectB
 Begin
  Var Struct AcctRec
   Begin
    Var Account1
    Var Account2
    Var Account3
End
Var TotalBalance
Proc DisplayTotalAccountBalance(ObjectA)
  If (System.Interrupt_Status <> 'Interrupt') And (System.Error_Status <>'Error') Then
    Call ObjectA.Read(AcctRec.Account1, AcctRec.Account2, AcctRec.Account3)
     Elif (System.Interrupt_Status = 'Interrupt) Then
      Call System_Interrupt()
    Else
     Call System_Error()
    Fi
 If (System.Interrupt_Status <> 'Interrupt') And (System.Error_Status <>'Error') Then
   TotalBalance := AcctRec.Account1 + AcctRec.Account2
   Elif (System.Interrupt_Status = 'Interrupt) Then
     Call System_Interrupt()
   Else
    Call System_Error()
   Fi
 If (System.Interrupt_Status <> 'Interrupt') And (System.Error_Status  <>'Error') Then
  TotalBalance := TotalBalance + AcctRec.Amount3
   Elif (System.Interrupt_Status = 'Interrupt) Then
   Call System_Interrupt()
   Else
   Call System_Error()
   Fi
If (System.Interrupt_Status <> 'Interrupt') And (System.Error_Status <>'Error') Then
  Put IOTerminal, TotalBalance
  Elif (System.Interrupt_Status = 'Interrupt) Then
   Call System_Interrupt()
  Else
```

```
    Call System_Error()
  Fi
End

Proc Main()
 Begin
  If (System.Interrupt_Status <> 'Interrupt') And (System.Error_Status <> 'Error) Then
   Call ObjectB.DisplayTotalAccountBalance(ObjectA)
Elif (System.Interrupt_Status = 'Interrupt') Then
  Call System_Interrupt()
Elif (System.Error_Status = 'Error') Then
  Call System_Error()
Fi
 End
```

D.8.2 C++ Representation:

```
#include<iostream.h>
#include<fstream.h>

Class ObjectA_Type {
 Public:
Struct FileRecord {
  Float Account1;
  Float Account2;
  Float Account3;
 }
Struct FileRecord_Index {
   Float Element1;
   Float Element2;
   Float Element3;
}
int FileRecord_Index_Count;
datetime CurDate;
Public:
Void ObjectA();
Void ~ObjectA();
Void ReadRec(float Account1, float Account2, float Account3);
} ObjectA;

Class ObjectB_Type {
 Public:
  Struct AcctRec {
    Float Account1;
    Float Account2;
    Float Account3;
  }
  sync float TotalBalance;
 Public:
 Void DisplayTotalBalance(ObjectA_Type Object);
} ObjectB;

main()
{
ObjectA = new ObjectA_Type;
ObjectB = new ObjectB_Type;
ObjectB.DisplayTotalBalance(ObjectA);
}

void ObjectA_Type::ObjectA_Type()
 {
try {
```

```
  this->CurDate=date;
  this->FileRecord = malloc(sizeof(this->FileRecord);
  this->FileRecord_Index_Count =1
}
catch { throw; } }

void ObjectA_Type::~ObjectA_Type()
 {
 try {
  free(this->AcctRec); }
catch { throw; }
}

void ObjectB_Type::ObjectB_Type()
 {
 try {
  this->AcctRec = malloc(sizeof(this->AcctRec)); }
catch {throw; }
}

void ObjectB_Type::~ObjectB_Type()
{
try {
  free(this->AcctRec); }
catch { throw; }
}

void ObjectA_Type::ReadRec(float Account1, float Account2, float Account3)
{
 try { fstream file1;
      file1.open('Acct.dat', ios::in) //Phys_Dev's value of FileRecord is substituted
// as the file path and the C++ equivalent of File_Mode of FileRecord is substituted
// for the file mode
       file1.seekg(this->FileRecord_Index_Count * sizeof(this->FileRecord), ios:beg);
       file1.read(this->FileRecord,sizeof(this->FileRecord));
       par {
            file1.close
            Account1 = this->FileRecord.Account1
            Account2 = this->FileRecord.Account2
            Account3 = this->FileRecord.Account3
}
 }
 catch { throw; }
}

void ObjectB_Type::DisplayTotalAccountBalance(ObjectA_Type Object)
{
try {
 Object.ReadRec(this->AcctRec.Account1, this->AcctRec.Account2, this->AcctRec.Account3);
TotalBalance = this->AcctRec.Account1 + this->AcctRec.Account2
TotalBalance = TotalBalance + this->AcctRec.Account3
Cout << TotalBalance
}
catch {throw; }
   }
```

D.8.3 Summary

This section demonstrates that translation from a typeless WSL to a C++ representation of a system is possible, provided that the original data typing of the source program is retained and then satisfactorily translated into C++ data types. Future work on TAGDUR is planned to further enhance this C++ code generation ability. With code generation, much development time and cost, as well as the possibility of errors in handling numerous variable names, is reduced over the traditional manual re-coding approach.

Appendix E: Transformation Experimental Data

E.1 Event Identification Experimental Data

The investigation focused on a 800 line COBOL program which was converted to a 9000 line WSL program. The thesis' event identification algorithm found a total of 374 potential interrupts within this program. The interrupts discovered were as follows:

Type of Event	Number of Events	Number of Asynchronous Events	Number of Synchronous Events	Number of Intra-Class * Events	Number of Inter-Class** Events
System	20	0	20	0	20
Error	20	0	20	0	20
File Read	19	0	19	0	19
File Write	202	136	66	0	202
Method Invokation	113	36	77	91	22

*** Intra-Class refers to a situation where events occur between methods of the same class**

**** Inter-Class refers to a situation where events occur between methods of different classes**

Inter-Class Method Invokations

Asych.	Synch	Task Differerence	Number
√		0 (Asynch)	59
	√	1	13
	√	2	8
	√	3	2
	√	4	2
	√	5	2
	√	6	2
	√	7	1
	√	8	1

Intra-Class Method Invokations

Asych.	Synch	Task Differerence	Number
√		0 (Asynch)	18
	√	1	4
	√	2	0
	√	3	0
	√	4	0
	√	5	0
	√	6	0
	√	7	0
	√	8	0

E.2 Results of Event Identification Experiments

If the degree of synchronicity within the system is examined, it can be determined that the degree of asynchronicity is highly dependent on the type of event. An example, File Read events are all synchronous. Visual inspection of the code reveals the reason for this synchronicity. The next task sequence group is dependent on the variables read by the File Read event for their execution. The File Write event has a more even distribution with the minority, 67%, being asynchronous with a degree of asynchronicity of one task sequence number with the remainder, 33%, being synchronous.

Error and system interrupts are all synchronous. Not only is there a data dependency but also a logic dependency – if a system or error interrupt occurs and until that interrupt is handled, there is no reason for normal execution of the program to continue.

In regards to method invocation events, intra-class (methods which are invoked from within their own class) events are 68% synchronous and 32% asynchronous with only one task sequence degree of asynchronicity. The

inter-class (methods which are invoked from outside their class) events are more evenly-distributed as to the degree of their asynchronicity. A majority of inter-class methods, 65%, are asynchronous, 9% are one task sequence degree of asynchronicity, 2% are two task sequence degree of asynchronicity, and so on until 8 task sequence degrees of asynchronicity is reached.

From these statistics it can be concluded that the object clustering methods, outlined in [6],which tried to cluster variables and procedures with data/control dependencies together, seems to have been successful. Methods which are invoked within the same class seem to have a higher degree of dependency than methods that are invoked outside their class.

The thesis' method of identifying events in legacy systems aids in the transformation of procedural code to event-based code, as part of the reengineering process of the legacy system.

Appendix F: WSL-UML Notations by Diagram Type

1. Class Diagram

VAR STRUCT Class [OMG, 2004: p 2-26]
BEGIN
VAR Type // always Class
VAR AttributesList // list of variables associated with this class
VAR ProcedureList // list of procedures, their parameters and implementations associated with this class
VAR Visibility // present in most components.
END
AttributesList: list of variables in the format VAR <VarName>
ProceduresList: list of procedures in the following format:
= Proc <ProcName>(<ParameterList>)
Begin // delimiter of procedural code
<WSL statements>
End.

ParameterList: list of variable names separated by commas
Var Struct Comment [OMG, 2004: p 2-30]
BEGIN
VAR Type // always Comment
VAR Comment
END

// links a comment to the UML model element that it refers to
VAR STRUCT CommentLink
BEGIN
VAR CommentName
VAR Type // always CommentLink
VAR RelatedUMLModelElementName
END

VAR STRUCT Association [OMG, 2004: p 2-19, 2-23]
BEGIN
VAR AssociationEndName1
VAR AssociationEndName2
VAR Type // always Association
VAR Persistance // how long does this association last? Is it permanent between objects or termporary
VAR Role // role that this association plays among its associated objects
VAR isXORConstraint // XORing of specified constraints upon thisobject
END

VAR STRUCT AssociationEnd [OMG, 2004: p 2-19, 2-23]
BEGIN
VAR Type // always AssociationEnd
VAR ObjectName // name of the object to which this association end is attached
VAR RoleName
VAR AggregationKind // compostion, aggregation
VAR Changeability // is this association end able to be changed, AddOnly, or is it frozen once it is defined?
Var Ordering // is this association end ordered? In this reengineering, it never is
Var IsNavigable // navigation
Var TargetScope // enumeration of instance or classifier
Var Visibility // enumeration of public, private, or protected
Var Multiplicity // can be one-to-one, one-to-many, zero-to-one, zero-to-many, et al
END

VAR STRUCT Package [OMG, 2004: p 2-6]
Begin
VAR PackageName // name of Package
VAR Type // always will be package
VAR AssociationList // list of associations that are included in this collection
Var ObjectNameList // list of objects that are included in this collection
Var Coupling
END

2. Object Diagram

VAR Object [OMG, 2004: p 3-34,3-45]
BEGIN
VAR Type // always Object
VAR Class // name of class that object is an instance of
VAR AttributesList // list of variables associated with this class
VAR ProcedureList // list of procedures, their parameters and implementations associated with this class
VAR Visibility // present in most components.
END

AttributesList: list of variables in the format VAR <VarName>
ProceduresList: list of procedures in the following format:
Proc <ProcName>(<ParameterList>)
Begin // delimiter of procedural code
<WSL statements>
End.

ParameterList: list of variable names separated by commas

Var Struct Comment [OMG, 2004: p 2-30]
BEGIN
VAR Type // always Comment
VAR Comment
END

// links a comment to the UML model element that it refers to
VAR STRUCT CommentLink
BEGIN
VAR CommentName
VAR Type // always CommentLink
VAR RelatedUMLModelElementName
END

VAR STRUCT Association [OMG, 2004: p 2-19, 2-23]
BEGIN
VAR AssociationEndName1
VAR AssociationEndName2
VAR Type // always Association
VAR Persistance // how long does this association last? Is it permanent between objects or termporary
VAR Role // role that this association plays among its associated objects
VAR isXORConstraint // XORing of specified constraints upon thisobject
END

VAR STRUCT AssociationEnd [OMG, 2004: p 2-19, 2-23]
BEGIN
VAR Type // always AssociationEnd
VAR ObjectName // name of the object to which this association end is attached
VAR RoleName
VAR AggregationKind // compostion, aggregation
VAR Changeability // is this association end able to be changed, AddOnly, or is it frozen once it is defined?
Var Ordering // is this association end ordered? In this reengineering, it never is
Var IsNavigable // navigation
Var TargetScope // enumeration of instance or classifier

Var Visibility // enumeration of public, private, or protected
Var Multiplicity // can be one-to-one, one-to-many, zero-to-one, zero-to-many, et al
END

VAR STRUCT Package [OMG, 2004: p 2-6]
Begin
VAR PackageName // name of Package
VAR Type // always will be package
VAR AssociationList // list of associations that are included in this collection
Var ObjectNameList // list of objects that are included in this collection
Var Coupling
END

3. Statechart Diagram

Var Struct State [OMG, 2004: pp 2-146, 2-147]
Begin
Var Type //always State
Var ActionLable
Var ActionExpression
Var Action // describes the action
Var ParentElement // StateVertex
Var Entry // procedure that is invoked when this state is entered
Var Exit //procedure that is invoked when this state is exited
Var InternalTransition // set of transitions that, if triggered, occur without exiting or entering this state and without causing a state change
Var DoActivity // optional procedure that is executed while being in this state
End

Var Struct Transition [OMG, 2004: pp 2-149,2-150]
Begin
Var Type // always Transition
Var ParentElement // always ModelElement
Var Trigger // specifies the event that fires the transition
Var Guard // a Boolean predicate that, when evaluated to true, governs the firing of the transition
Var Effect // specifis an optional procedure to be performed when the transition fires
Var Source //designates the originating state vertex, state, or pseudostate, of the transition
Var Target //designates the target state vertex, state, or pseudostate that is reached when the transition is taken
End

Var Struct Guard [OMG, 2004: pp 2-144-145]
Begin
Var Type //always Guard
Var Expression //Boolean expression that specified the guard
Var ParentElement // always ModelElement
End

Var Struct SignalEvent
Begin
Var Type //always SignalEvent
Var Signal // specified signal that is associated with this event
Var ParentElement // always Event
End

Var Struct StateMachine [OMG, 2004: 2-147, 2-148]
Begin
Var Type // always StateMachine
Var Context // an association to the model element whose behaviour is specified by this statemachine
Var Top //designates the top-level state that is the root of the state containment hierarchy
Var Transition // a list of transitions that belong to this state machine
Var ParentElement // always ModelElement

End

Var Struct SubmachineState [OMG, 2004: pp 2-148-149]
Begin
Var Type // always SubmachineState
Var ParentElement //always Composite State
Var Submachine // name of the state machine that is substituted in place of the submachine state.
End

4. Sequence/Collaboration Interaction Diagram

VAR SequenceDiagramPackage – a list of tuples (Sequence, Message, TargetObject, SourceObject) – a package enclosing the entire sequence diagram that contains the objects involved in this diagram along with their lifelines, sequences, and messages.
VAR STRUCT Sequence [OMG, 2004: p 2-115, 2-119]
VAR Type // always Sequence
VAR SwimlaneName // each class object is assigned its own swimlance
VAR InvokingObject
VAR InvokingProcedureName
VAR TargetObject
VAR TargetProcedureName
VAR ProcedureParameterList // list of parameters passed by the procedure
VAR SequenceNumber // the sequence to which the procedure may be invoked. If procedures may execute in parallel, these sequence numbers are the same.
END

VAR STRUCT Message [OMG, 2004: p 2-115, 2-119]
Begin
Var Type // always Message
Var MessageName
Var Asynchronous
Var TargetObjectName
Var SourceObjectName
Var MessageSequenceNumber
Var Condition
Var IterationMarket
Var TargetProcedureNameCalled
End

VAR STRUCT Partition [OMG, 2004: p 2-115, 2-119]
Begin
VAR Type // Always Partition
VAR ObjectName // name of object in partition
VAR MessageList // list of messages interacting with objects in partition
VAR SequenceList // list of sequences interacting with objects in partition
End
Var PartitionList – list of all partitions in sequence diagram

5. Activity Diagram

VAR STRUCT Activity
BEGIN
VAR Type //always Activity
VAR ActionState // is the activity an activity or action state
VAR Description // description of the activity
VAR CodeLine
VAR OperationType //read, write, method invocation, assignment
VAR SourceVariable
VAR DestinationVariable
VAR IndexVariable
VAR StartActivity // is this activity the start activity of the diagram
VAR EndActivity // is this acitivity the end activity of the diagram
END

VAR STRUCT ActionState [OMG, 2004: pp 2-172]
Begin
VAR Type // always ActionState
VAR EntryAction //specifies the entry action association
VAR Description // description of the activity
VAR CodeLine
VAR OperationType //read, write, method invocation, assignment
VAR SourceVariable
VAR DestinationVariable
VAR IndexVariable
VAR StartActivity // is this activity the start activity of the diagram
VAR EndActivity // is this acitivity the end activity of the diagram
VAR DynamicArguments // an ArgLists expression that determines the number of parallel executions of the actions of the state
VAR DynamicMultiplicity // a multiplicity that constrains the number of parallel executions of the actions of state
VAR IsDynamic // a Boolean value specifying whether the state's actions might be executed concurrently
End

VAR STRUCT Transition [OMG, 2004: p 2-175]
BEGIN
VAR Type // Always Transition
VAR IncomingActivitiesList // list of activities coming into this transition, by activityname
VAR OutgoingActivitiesList // list of activities going out of this transition, by activityname
VAR TransitionType // simple, fork, join
VAR Effect
VAR Guard // condition under which this transition must satisfy before it is fired
VAR Iteration // iteration condition
END

VAR STRUCT Event [OMG, 2004: p 2-142, 2-144]
BEGIN
VAR Type // always Event
VAR TargetObject // object receiving the event
VAR TargetOperation // procedure being invoked by the event, if relevant
VAR TargetComponent // name of target component – state or activity
VAR SourceObject //object invoking the event
VAR SourceOperation //procedure where event was invoked or occurred
VAR SourceComponent // name of source component – state or activity
VAR Asynchronous // is event asynchronous or synchronous
VAR EventType // type of event
VAR EventDescription //description of the event
VAR Condition // condition or guard that must be satisfied before event occurs
VAR EventParameterList // list of parameters that the event passes to the target object
END

6. Component Diagram

VAR STRUCT Component [OMG, 2004: p 2-31]
BEGIN
VAR Type // always Component
VAR ComponentName
END

VAR STRUCT ComponentDeploymentLocation [OMG, 2004: p 2-31]
Begin
Var Type // always DeploymentLocation
Var DeploymentLocation // set of Nodes that the Component resides on
End

Var Struct Resident [OMG, 2004: p 2-31]
Begin

VAR Type // always resident
VAR ResidentList – list of model elements that specify the component
END

Var Struct Implementation [OMG, 2004: p 2-31]
Begin
Var Type //always Implementation
Var ImplementationList – set of Artefacts that implement the Component
End
Var ComponentList – list of ordered pairs of Component, Resident, and Implementation. // this list represents the Components (software libraries, copybooks, etc) in a comma-delimited paired form

7. Deployment Diagram

VAR STRUCT Deployment
BEGIN
VAR Type // always Deployment
VAR SuperType
VAR LogicalDeviceName // when a device has a logical name (Gemini)
VAR PhysicalDeviceName // when a device has a physical name 1..22.44.32
END

VAR STRUCT DeploymentLink
BEGIN
VAR Type // always DeploymentLink
VAR DeploymentEnd1Name // corresponds to the deployment structure at one end
VAR DeploymentEnd2Name // corresponds to the deployment structure at the other end
END
VAR DeploymentList – list of Deployments in Deployment Diagram

8. Use Case Diagram

Diagram is represented by:
UseCaseList – a list of tuples (UseCase, UseCaseExtend, UseCaseAssociation)
Var STRUCT UseCase [OMG, 2004: pp 2-129, 2-133]
Begin
Var Type // always UseCase
Var ActorList // list of actors involved in this use case
Var UseCaseDescription //description of use case
Var Condition // condition under which this use case applies
END

Var Struct Actor [OMG, 2004: pp 2-129, 2-133]
Begin
Var ActorName
Var Type // always Actor
Var Role
END

Var Struct UseCaseExtend [OMG, 2004: pp 2-129, 2-133]
Begin
Var Condition
End
Var Struct UseCaseAssociation [OMG, 2004: pp 2-129, 2-133]
Begin
Var Type // always UseCaseAssociation
Var Base // use case to be extended
Var Extension // use case specifying the extending behaviour
END

Appendix G: Comments by Local Supervisor

G.1 Short Biography

The local supervisor, Dave Shellenberg, is an experienced Information Technology professional. He has over 25 years of Information Technology experience, including 21 years in the particular telecommunications company. Consequently, he is both experienced and very familiar with this company's systems, including main-frame batch-oriented systems such as the selected legacy system used in this thesis. Mr. Shellenberg also has expertise in both COBOL and UML. Because of his extensive experience and expertise, Mr Shellenberg is able to expertly comment on the COBOL to WSL translation, proposed restructuring, and UML diagram extraction processes. Because the research result of this thesis has a large amount of applicability to the information technology industry, it was fortunate that Mr Shellenberg agreed to act as the local supervisor and to lend his many years of industry experience and expertise in both evaluating the thesis' approach and in providing an industry viewpoint as to these results.

G.2 Mr. Shellenberg's Comments on Thesis' Proposed Approach and Validation of TADUR's Output

Although Mr. Shellenberg acted as the local supervisor, he evaluated the thesis' approach and results in an objective and professional way.

Mr. Shellenberg recognized, from his personal experience at the company, the thesis' selected legacy system as part of the telecommunication company's core set of systems, handling Customer Service Orders. This system was a batch-oriented COBOL system that ran on an IBM 370 mainframe computer at one of the telecommunication's data centres.

As the letter in Chapter 11 states, Mr. Shellenberg recognised that the COBOL to WSL conversion process appears to effectively represent significant portions of the original source code and that the WSL to UML notation conversion process models the processes, relationships, and structures of the WSL representation and of the original COBOL code.

Mr. Shellenberg, from his industry experience, was able to comment that because there still exist many procedural legacy applications, a validated method of assisting in the reengineering of these applications to object-orientated implementations could be financially beneficial to the industry.

G.3 Mr. Shellenberg's Inability to Comment Further on TAGDUR's Output or Usefulness

In his later emails (late 2005), Mr. Shellenberg commented:"I am now working on two projects (at the same time). Due to my busy schedule, I barely have time to check my email during the week with all that's happening for me. Right now, I'm sorry but I can't provide any additional feedback for you". Consequently, Mr. Shellenberg was unavailable to validate the UML diagrams produced from the selected sample.

Appendix H: UML Checklist (Validation of Generated UML Diagrams)

The validation that this checklist provides is for the validation of UML diagrams produced from object-oriented WSL code. The original COBOL code was not used for several reasons:

1) the COBOL code is procedurally-structured and, therefore, it is impossible to model in UML
2) many constructs in COBOL, such as CONTINUE, are not modelled in WSL
3) additional variables in WSL, such as index variables are not used in COBOL

There is a need to focus on what needs to be validated:

1) sequence diagram – one diagram
2) activity diagram – diagram by procedure
3) component diagram – one diagram

1) Class Diagram

Are all WSL variables and procedures in classes as attributes and methods respectively (principle of inclusion)?
Are any non-variables or non-procedures included?
For each method, do communication paths exist that provide access to necessary attribute data located in other classes?

Variable Name	**Owning Class**	**Present in Diagram**
UT-S-URM	Class10	√
UT-S-USOC	Class10	√
100-URM-FILE	Class9	√
100-URM-FILE.ELEMENT1.ELEMENT1	Class9	√
100-URM-FILE.ELEMENT2	Class 9	√
100-URM-FILE.ELEMENT3	Class9	√
100-URM-FILE.ELEMENT4	Class9	√
100-URM-FILE.ELEMENT5	Class9	√
100-URM-FILE.ELEMENT6.ELEMENT1	Class9	√
100-URM-FILE.ELEMENT6.ELEMENT1	Class9	√
100-URM-FILE.ELEMENT6.ELEMENT2	Class9	√
100-URM-FILE.ELEMENT6.ELEMENT1	Class9	√

100-URM-FILE.ELEMENT6.ELEMENT3	Class9	√
100-URM-FILE.ELEMENT6.ELEMENT4	Class9	√
100-URM-FILE.ELEMENT6.ELEMENT5	Class9	√
100-URM-FILE.ELEMENT6.ELEMENT6	Class9	√
100-URM-FILE.ELEMENT6.ELEMENT7	Class9	√
100-URM-FILE.ELEMENT7	Class9	√
100-URM-FILE.INDEXELEMENT1	Class9	√
100-URM-FILE.INDEXELEMENT2	Class9	√
100-URM-FILE.INDEXELEMENT3	Class9	√
100-URM-FILE.INDEXELEMENT4	Class9	√
100-URM-FILE.INDEXELEMENT5	Class9	√
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT1	Class9	√
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT2	Class9	√
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT3	Class9	√
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT4	Class9	√
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT5	Class9	√
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT6	Class9	v
100-URM-FILE.INDEXELEMENT6.INDEXELEMENT7	Class9	√
100-URM-FILE.INDEXELEMENT7	Class9	√
100-URM-REC	Class11	√
200-USOC-FILE	Class11	√
200-USOC-REC.ELEMENT1	Class10	√
200-USOC-REC.ELEMENT2	Class10	√
200-USOC-REC.INDEXELEMENT1	Class10	√
200-USOC-REC.INDEXELEMENT2	Class10	√
1000-URM-DETAIL.1000-URM-USOC.1000-URM-USOC	Class9	√
1000-URM-DETAIL.1000-URM-CLASS	Class9	√
1000-URM-DETAIL.1000-URM-COS-DESCRIPTION	Class9	√
1000-URM-DETAIL..1000-URM-USOC-DESCRIPTION	Class9	√
1000-URM-DETAIL.1000-URM-USOC-DESCRIPTION	Class9	√
1000-URM-DETAIL.1000-URM-PREPRATE-IND	Class9	√
1000-URM-DETAIL.1000-USOC-BY-RG[1].1000-USOC-BY-RG	Class9	√
1000-URM-DETAIL.1000-USOC-BY-RG[2].1000-USOC-BY-RG	Class9	√
1000-URM-DETAIL.1000-USOC-BY-RG[3].1000-USOC-BY-RG	Class9	√
1000-URM-DETAIL.1000-USOC-BY-RG[4].1000-USOC-BY-RG	Class9	√
1000-URM-DETAIL.1000-USOC-BY-RG[5].1000-USOC-BY-RG	Class9	√

1000-URM-DETAIL.1000-USOC-BY-RG[6].1000-USOC-BY-RG	Class9	√
1000-URM-DETAIL.1000-USOC-BY-RG[7].1000-USOC-BY-RG	Class9	√
1000-URM-DETAIL.FILLER1	Class9	√
1100-MISC-FIELDS.1100-LAST-USOC	Class8	√
1100-MISC-FIELDS.1100-CLASS-SERVICE.1100-BUS-COM-CLASS	Class8	√
1400-FILE-STATUS.1400-URM-STATUS.1400-URM-STATUS-OK	Controller	√
1400-FILE-STATUS.1400-URM-STATUS.1400-URM-EOF	Controller	√
1400-FILE-STATUS.1400-USOC-STATUS.1400-USOC-STATUS-OK	Controller	√
2000-USOC-DETAIL.2000-USOC	Class10	√
2000-USOC-DETAIL.FILLER1	Class10	√
2000-USOC-DETAIL.INDEXELEMENT1	Class10	√
2000-USOC-DETAIL.INDEXELEMENT2	Class10	√
9900-ABEND-FIELD.9900-STAT-OPCODE	Class12	√
9900-ABEND-FIELD.9900-FILENAME	Class12	√
9900-ABEND-FIELD.9900-PARANAME	Class12	√
9900-ABEND-FIELD.9900-STAT-PROGNAME	Class12	√
9900-ABEND-FIELD.9900-STAT-CODE	Class12	√
9900-ABEND-FIELD.9900-STAT-ERROR-KEY	Class 12	√

Example of Associations

WSL Codeline	**Owning Class**	**Class Being Accessed**	**Class Association Modelled?**
100-URM-FILE.PHYS_DEV := 'UT-S-URM'	Controller	Class11	√
9900-ABEND-FIELD.9900-STAT-PROGNAME := 'CR750'	Controller	ExternalClass11	√
CALL 0000-CREATE-NETWORK-USOC-TABLE( )	Controller	Class9	√
WHILE NOT 1000-URM-DETAIL.1000-URM-USOC.1000-URM-TRAILER DO	Class8	Class9	√
CALL 1100-OPEN-FILES( )	Class8	Class11	√
CALL 9000-READ-URM( )	Class8	Class9	√
9900-ABEND-FIELD.9900-STAT-OPCODE := 'OPEN '	Class11	ExternalClass12	√
IF NOT (1400-FILE-STATUS.1400-USOC-STATUS.1400-USOC-STATUS-OK) THEN	Class11	Controller	√
IF 1000-URM-DETAIL.1000-URM-RATES[1].1000-USOC-BY-RG <> 1000-URM-DETAIL.1000-URM-RATES[2].1000-USOC-BY-RG AND 1000-URM-DETAIL.1000-URM-RATES[1].1000-USOC-BY-RG > 0 AND 1000-URM-DETAIL.1000-URM-USOC <> 1100-MISC-FIELDS.1100-LAST-USOC AND 1100-MISC-FIELDS.1100-CLASS-SERVICE.1100-BUS-COM-CLASS THEN	Class 8	Class9	√
2000-USOC-DETAIL.2000-USOC := 1100-MISC-FIELDS.1100-LAST-USOC	Class8	Class10	√

100-URM-FILE.ISOPEN := 'FALSE'	Class8	Class11	√
9900-ABEND-FIELDS.9900-PARANAME := '3000-TERMINATE'	Class8	ExternalClass12	√
1400-FILE-STATUS.1400-URM-STATUS.1400-URM-STATUS-OK	Class8	Controller	√
IF NOT 1400-FILE-STATUS.1400-URM-STATUS.1400-URM-STATUS-OK THEN	Class9	Controller	√
9900-ABEND-FIELD.9900-PARANAME := '9000-READ-URM'	Class9	ExternalClass12	√
IF NOT 1400-FILE-STATUS.1400-USOC-STATUS.1400-USOC-STATUS-OK THEN	Class10	Controller	√
9900-ABEND-FIELD.9900-STAT-CODE := 1400-USOC-STATUS	Class 10	ExternalClass12	√

2) Sequence Diagram(s)

- For each class, are all file I/O calls modelled as calls between classes where they originated and the File object?
- For each class, is there at least one system interrupt/error invocation call (their originating object) to the System object?
- If an inter-class procedure call is made, does it include a call between the caller class, the originating class, and the called class, the destination class?
- If control constructs surround a call/event, are they included as guards?
  If nested constructs are the guards in the format [<outerguard> AND <innerguard>]?
  If a control construct is a condition, is it in the format [<condition>]?
  If a control construct is an iteration condition, is it in the format [<condition>}*?
- For an event with parameters, are all and no extra parameters included in the parameter list of the sequence call in the format [<condition>]/Call(param1, param2,…)
- Are any intra-class procedure calls included in the calls modelled? They should not be.

Codeline	Guard	Invoking Class	Invoked Class	Present
CALL 0000-CREATE-NETWORK-USOC-TABLE( )	N/a	Controller	Class8	√
CALL 1000-INITIALIZE( )		Class8	Class8	N/a (intra-class)
CALL 2000-PROCESS-URM( )	[NOT (1000-URM-DETAIL.1000-URM-USOC.1000-URM-TRAILER)]*	Class8	Class8	N/a (intra-class)
CALL 3000-TERMINATE( )		Class8	Class8	N/a (intra-class)
CALL 1100-OPEN-FILES( )		Class8	Class11	√
CALL 9000-READ-URM( )		Class8	Class9	√
CALL U076(9900-ABEND-FIELD)	[NOT (1400-FILE-STATUS.1400-URM-STATUS.1400-URM-STATUS-OK)]	Class11	ExternalClass12	√
CALL U076(9900-ABEND-FIELD)	[NOT (1400-FILE-STATUS.1400-USOC-STATUS.1400-USOC-STATUS-OK)]	Class11	ExternalClass12	√
CALL 9100-WRITE-USOC()	1000-URM-DETAIL.1000-URM-RATES[1].1000-USOC-BY-RG <> 1000-URM-DETAIL.1000-URM-RATES[2].1000-USOC-BY-RG	Class8	Class10	√

	AND 1000-URM-DETAIL.1000-URM-RATES[1].1000-USOC-BY-RG > 0 AND 1000-URM-DETAIL.1000-URM-USOC <> 1100-MISC-FIELDS.1100-LAST-USOC AND 1100-MISC-FIELDS.1100-CLASS-SERVICE.1100-BUS-COM-CLASS			
CALL 9000-READ-URM()		Class8	Class9	√
CALL U076(9900-ABEND-FIELD)	[NOT (1400-FILE-STATUS.1400-URM-STATUS.1400-URM-STATUS-OK)]	Class8	ExternalClass12	√
CALL U076(9900-ABEND-FIELD)	[NOT (1400-FILE-STATUS.1400-USOC-STATUS.1400-USOC-STATUS-OK)]	Class8	ExternalClass12	√
FETCH 1000-URM-DETAIL.1000-URM-USOC.1000-URM-TRAILER , 100-URM-FILE.ELEMENT1.ELEMENT1, 100-URM-FILE.INDEXELEMENT1.INDEXELEMENT1		Class9	File	√
FETCH 1000-URM-DETAIL.1000-URM-CLASS , 100-URM-FILE.ELEMENT2 , 100-URM-FILE.INDEXELEMENT2		Class9	File	√
FETCH 1000-URM-DETAIL.1000-URM-COS-DESCRIPTION , 100-URM-FILE.ELEMENT3 , 100-URM-FILE.INDEXELEMENT3		Class9	File	√
FETCH 1000-URM-DETAIL.1000-URM-USOC-DESCRIPTION , 100-URM-FILE.ELEMENT4 , 100-URM-FILE.INDEXELEMENT4		Class9	File	√
FETCH 1000-URM-DETAIL.1000-URM-CURRENT-PRERATE-IND , 100-URM-FILE.ELEMENT5 , 100-URM-FILE.INDEXELEMENT5		Class9	File	√
FETCH 1000-URM-DETAIL.1000-URM-RATES[1].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT1 , 100-URM-FILE.INDEXELEMENT6.INDEXELEMENT1		Class9	File	√

FETCH 1000-URM-DETAIL.1000-URM-RATES[2].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT2 , 100-URM-FILE.INDEXELEMENT6.INDEXELEMENT2		Class9	File	√
FETCH 1000-URM-DETAIL.1000-URM-RATES[3].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT3 , 100-URM-FILE.INDEXELEMENT6.INDEXELEMENT3		Class9	File	√
FETCH 1000-URM-DETAIL.1000-URM-RATES[4].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT4 , 100-URM-FILE.INDEXELEMENT6.INDEXELEMENT4		Class9	File	√
FETCH 1000-URM-DETAIL.1000-URM-RATES[5].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT5 , 100-URM-FILE.INDEXELEMENT6.INDEXELEMENT5		Class9	File	√
FETCH 1000-URM-DETAIL.1000-URM-RATES[6].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT6 , 100-URM-FILE.INDEXELEMENT6.INDEXELEMENT6		Class9	File	√
FETCH 1000-URM-DETAIL.1000-URM-RATES[7].1000-USOC-BY-RG , 100-URM-FILE.ELEMENT6.ELEMENT7 , 100-URM-FILE.INDEXELEMENT6.INDEXELEMENT7		Class9	File	√
FETCH 1000-URM-DETAIL.FILLER1 , 100-URM-FILE.ELEMENT7 , 100-URM-FILE.INDEXELEMENT7		Class9	File	√
CALL U076(9900-ABEND-FIELD)	[NOT 1400-FILE-STATUS.1400-URM-STATUS.1400-URM-STATUS-OK]	Class9	ExternalClass12	√
PUT 200-USOC-REC.ELEMENT1, 2000-USOC-DETAIL.2000-USOC , 2000-USOC-DETAIL.INDEXELEMENT1		Class10	File	√
PUT 200-USOC-REC.ELEMENT2,		Class10	File	√

2000-USOC-DETAIL.FILLER1 , 2000-USOC-DETAIL.INDEXELEMENT2				
CALL U076(9900-ABEND-FIELD)	[NOT 1400-FILE-STATUS.1400-USOC-STATUS.1400-USOC-STATUS-OK]	Class10	ExternalClass12	√

3) Component Diagram(s)

- for each software module,
  determine any included libraries or source files called from within the software module by locating the included libraries/source file using the format "include <filename>"
  -find any logical file/record to physical mapping
  -if <logical_filename>.DataRecord := <LogicalRecordName> is encountered, is it modeled as an association between the logical record and the logical filename?
  -if <logical_filename>.PhysDev := <Physical File> is encountered, is it modelled as an association between the logical filename to the physical device?

Component	Association Modelled	CheckMark
Included Library files in the format include <File Name>. None found.	None	√
Physical file to logical file mapping: 'UT-S-URM' physical device to logical file name, 100-URM-FILE	Modelled as physical device to logical file mapping in component diagram	√
Logical file name, 100-URM-FILE, to data record, 100-URM-REC	Modelled as logical file name to data record in component diagram	√
Physical file to logical file mapping: 'UT-S-USOC' physical device to logical file name, 200-USOC-FILE	Modelled as physical device to logical file mapping in component diagram	√
Logical file name, 200-USOC-FILE, to data record, 200-USOC-REC	Modelled as logical file name to data record in component diagram	√

4) Deployment Diagram

- for each software module,
  - if any keywords indicate a physical device (such as IOTerminal, IOKeyword, etc), is the relationship between the identified physical device and the software module that is being parsed modelled?

Physical Device	Relationship Modelled	CheckMark
No physical devices (other than file systems – which are mentioned in the Component Diagram) are found in the WSL code	N/A	N/A

5) Activity Diagrams

for each WSL codeline (other than control constructs, or procedure headers, variable declaration, or keywords such as Begin or End) is an action state/activity modelled?
For any control constructs
do each guard, in the format [<Condition>], have an exit condition, in the format [<Not Condition>], and a state to transition to even if this state is the end state of the diagram?
If nested constructs are the guards in the format [<outerguard> AND <innerguard>]?
If a control construct is a condition, is it in the format [<condition>]?
If a control construct is an iteration condition, is it in the format [<condition>}*?

Are actions that can be executed in parallel modelled as parallel flows?
Are actions that can be executed in sequential flow modelled as sequential flows?
If two or more states that are in parallel flows succeed to a sequential executing successive states, the preceding states in parallel flows must transition to a merge synchronisation bar and then from this bar, a transition to the sequential successive state. Is this modelled in all of these cases?
If a sequentially-executing state transitions to successive states that can execute in parallel, the preceding, sequentially-executing states must transition to a synchronisation fork bar. From this bar, a transition from the bar to parallel-executing successive states which are modelled as parallel flows. Is this modelled in all of these cases?
Each procedure forms its own diagram
Does each diagram have an appropriate start and end state?
Each state, other than start or end state, represents a WSL codeline. Does each state, other than the start and end states, have at least one transition to and at least one transition from?

Method	**Criteria**	**Example**	**Present**
CR705V02_VAR-INIT( )	Is each state, other than those of control constructs and those consolidated into a single file opertion state such as multiple record field I/O access states for the same record , modelled?		√
	Is each guard modelled and in the proper format? []: condition; []* iteration?	N/a	√
	Does each guard have a non-guard exit transition?	N/a	√
	Are actions that can be executed in parallel modelled as parallel flows?	N/a	√
	Are actions that can be executed in sequential flow modelled as sequential flows?	N/a	√
	If two or more states that are in parallel flows succeed to a sequential executing successive states, the preceding states in parallel flows must transition to a merge synchronisation bar and then from this bar, a transition to the sequential successive state. Is this modelled in all of these cases?		√
	If a sequentially-executing state transitions to successive states that can execute in parallel, the preceding, sequentially-executing states must transition to a synchronisation fork bar. From this bar, a transition from the bar to parallel-executing successive states which are modelled as parallel flows. Is this modelled in all of these cases?		√
	Does each diagram have an appropriate start and end state?		√
	Each state, other than start or end state, represents a WSL codeline. Does each state, other than the start and end states, have at least one transition to and at least one transition from?		√
0000-CREATE-NETWORK-USOC-TABLE( )	Is each state, other than those of control constructs and those consolidated into a single file opertion state such as multiple record field I/O access states for the same record , modelled?		√
	Is each guard modelled and in the proper format? []: condition; []* iteration?	[NOT 1000-URM-DETAIL.1000-URM-USOC.1000-URM-TRAILER]*	√
	Does each guard have a non-guard exit transition?	[1000-URM-DETAIL.1000-URM-USOC.1000-URM-TRAILER]/CALL 3000-TERMINATE( )	√

Method	Criteria	Example	Present
	Are actions that can be executed in parallel modelled as parallel flows?		√
	Are actions that can be executed in sequential flow modelled as sequential flows?		√
	If two or more states that are in parallel flows succeed to a sequential executing successive states, the preceding states in parallel flows must transition to a merge synchronisation bar and then from this bar, a transition to the sequential successive state. Is this modelled in all of these cases?		√
	If a sequentially-executing state transitions to successive states that can execute in parallel, the preceding, sequentially-executing states must transition to a synchronisation fork bar. From this bar, a transition from the bar to parallel-executing successive states which are modelled as parallel flows. Is this modelled in all of these cases?		√
	Does each diagram have an appropriate start and end state?		√
	Each state, other than start or end state, represents a WSL codeline. Does each state, other than the start and end states, have at least one transition to and at least one transition from?		√
1000-INITIALIZE( )	Is each state, other than those of control constructs and those consolidated into a single file opertion state such as multiple record field I/O access states for the same record , modelled?		√
	Is each guard modelled and in the proper format? []: condition; []* iteration	N/a	√
	Does each guard have a non-guard exit transition?	N/a	√
	Are actions that can be executed in parallel modelled as parallel flows?		√
	Are actions that can be executed in sequential flow modelled as sequential flows?		√
	If two or more states that are in parallel flows succeed to a sequential executing successive states, the preceding states in parallel flows must transition to a merge synchronisation bar and then from this bar, a transition to the sequential successive state. Is this modelled in all of these cases?		√
	If a sequentially-executing state transitions to successive states that can execute in parallel, the preceding, sequentially-executing states must transition to a synchronisation fork bar. From this bar, a transition from the bar to parallel-executing successive states which are modelled as parallel flows. Is this modelled in all of these cases?		√
	Does each diagram have an appropriate start and end state?		√
	Each state, other than start or end state, represents a WSL codeline. Does each state, other than the start and end states, have at least one transition to and at least one transition from?		√

Method	Criteria	Example	Present
1100-OPEN-FILES	Is each state, other than those of control constructs and those consolidated into a single file opertion state such as multiple record field I/O access states for the same record , modelled?		√
	Is each guard modelled and in the proper format? []: condition; []* iteration?	[NOT (1400-FILE-STATUS.1400-URM-STATUS.1400-URM-STATUS-OK)];[NOT (1400-FILE-STATUS.1400-USOC-STATUS.1400-USOC-STATUS-OK)]	√
	Does each guard have a non-guard exit transition?	[(1400-FILE-STATUS.1400-URM-STATUS.1400-URM-STATUS-OK)];[(1400-FILE-STATUS.1400-USOC-STATUS.1400-USOC-STATUS-OK)]	√
	Are actions that can be executed in parallel modelled as parallel flows?		√
	Are actions that can be executed in sequential flow modelled as sequential flows?		√
	If two or more states that are in parallel flows succeed to a sequential executing successive states, the preceding states in parallel flows must transition to a merge synchronisation bar and then from this bar, a transition to the sequential successive state. Is this modelled in all of these cases?		√
	If a sequentially-executing state transitions to successive states that can execute in parallel, the preceding, sequentially-executing states must transition to a synchronisation fork bar. From this bar, a transition from the bar to parallel-executing successive states which are modelled as parallel flows. Is this modelled in all of these cases?		v
	Does each diagram have an appropriate start and end state?		√
	Each state, other than start or end state, represents a WSL codeline. Does each state, other than the start and end states, have at least one transition to and at least one transition from?		√
2000-PROCESS-URM	Is each state, other than those of control constructs and those consolidated into a single file opertion state such as multiple record field I/O access states for the same record , modelled?		√

Method	Criteria	Example	Present
	Is each guard modelled and in the proper format? []: condition; []* iteration?	[IF 1000-URM-DETAIL.1000-URM-RATES[1]. 1000-USOC-BY-RG <> 1000-URM-DETAIL.1000-URM-RATES[2]. 1000-USOC-BY-RG AND 1000-URM-DETAIL.1000-URM-RATES[1]. 1000-USOC-BY-RG > 0 AND 1000-URM-DETAIL.1000-URM-USOC <> 1100-MISC-FIELDS.1100-LAST-USOC AND 1100-MISC-FIELDS.1100-CLASS-SERVIC E.1100-BUS-COM-CLASS THEN 1100-MISC-FIELDS.1100-LAST-USOC := 1000-URM-DETAIL.1000-URM-USOC]	√
	Does each guard have a non-guard exit transition?	[NOT IF 1000-URM-DETAIL.1000-URM-RATES[1]. 1000-USOC-BY-RG <> 1000-URM-DETAIL.1000-URM-RATES[2]. 1000-USOC-BY-RG AND 1000-URM-DETAIL.1000-URM-RATES[1]. 1000-USOC-BY-RG > 0 AND 1000-URM-DETAIL.1000-URM-USOC <> 1100-MISC-FIELDS.1100-LAST-USOC AND 1100-MISC-FIELDS.1100-CLASS-SERVIC E.1100-BUS-COM-CLASS THEN 1100-MISC-FIELDS.1100-LAST-USOC := 1000-URM-DETAIL.1000-URM-USOC]	√
	Are actions that can be executed in parallel modelled as parallel flows?		√
	Are actions that can be executed in sequential flow modelled as sequential flows?		√

Method	Criteria	Example	Present
	If two or more states that are in parallel flows succeed to a sequential executing successive states, the preceding states in parallel flows must transition to a merge synchronisation bar and then from this bar, a transition to the sequential successive state. Is this modelled in all of these cases?		√
	If a sequentially-executing state transitions to successive states that can execute in parallel, the preceding, sequentially-executing states must transition to a synchronisation fork bar. From this bar, a transition from the bar to parallel-executing successive states which are modelled as parallel flows. Is this modelled in all of these cases?		√
	Does each diagram have an appropriate start and end state?		√
	Each state, other than start or end state, represents a WSL codeline. Does each state, other than the start and end states, have at least one transition to and at least one transition from?		√
3000-TERMINATE	Is each state, other than those of control constructs and those consolidated into a single file opertion state such as multiple record field I/O access states for the same record , modelled?		√
	Is each guard modelled and in the proper format? []: condition; []* iteration?	[NOT (1400-FILE-STATUS.1400-URM-STATUS.1400-URM-STATUS-OK)];[NOT (1400-FILE-STATUS.1400-USOC-STATUS.1400-USOC-STATUS-OK)]	√
	Does each guard have a non-guard exit transition?	[(1400-FILE-STATUS.1400-URM-STATUS.1400-URM-STATUS-OK)];[(1400-FILE-STATUS.1400-USOC-STATUS.1400-USOC-STATUS-OK)]	√
	Are actions that can be executed in parallel modelled as parallel flows?		√
	Are actions that can be executed in sequential flow modelled as sequential flows?		√
	If two or more states that are in parallel flows succeed to a sequential executing successive states, the preceding states in parallel flows must transition to a merge synchronisation bar and then from this bar, a transition to the sequential successive state. Is this modelled in all of these cases?		√
	If a sequentially-executing state transitions to successive states that can execute in parallel, the preceding, sequentially-executing states must transition to a synchronisation fork bar. From this bar, a transition from the bar to parallel-executing successive states which are modelled as parallel flows. Is this modelled in all of these cases?		√
	Does each diagram have an appropriate start and end state?		√

Method	Criteria	Example	Present
	Each state, other than start or end state, represents a WSL codeline. Does each state, other than the start and end states, have at least one transition to and at least one transition from?		√
9000-READ-URM	Is each state, other than those of control constructs and those consolidated into a single file opertion state such as multiple record field I/O access states for the same record , modelled?		√
	Is each guard modelled and in the proper format? []: condition; []* iteration?	[NOT 1400-FILE-STATUS.1400-URM-STATUS.1400-URM-STATUS-OK]	√
	Does each guard have a non-guard exit transition?	[1400-FILE-STATUS.1400-URM-STATUS.1400-URM-STATUS-OK]	√
	Are actions that can be executed in parallel modelled as parallel flows?		√
	Are actions that can be executed in sequential flow modelled as sequential flows?		V
	If two or more states that are in parallel flows succeed to a sequential executing successive states, the preceding states in parallel flows must transition to a merge synchronisation bar and then from this bar, a transition to the sequential successive state. Is this modelled in all of these cases?		√
	If a sequentially-executing state transitions to successive states that can execute in parallel, the preceding, sequentially-executing states must transition to a synchronisation fork bar. From this bar, a transition from the bar to parallel-executing successive states which are modelled as parallel flows. Is this modelled in all of these cases?		√
	Does each diagram have an appropriate start and end state?		√
	Each state, other than start or end state, represents a WSL codeline. Does each state, other than the start and end states, have at least one transition to and at least one transition from?		√
9100-WRITE-USOC	Is each state, other than those of control constructs and those consolidated into a single file opertion state such as multiple record field I/O access states for the same record , modelled?		√
	Is each guard modelled and in the proper format? []: condition; []* iteration?	[NOT 1400-FILE-STATUS.1400-USOC-STATUS.1400-USOC-STATUS-OK]	√
	Does each guard have a non-guard exit transition?	[1400-FILE-STATUS.1400-USOC-STATUS.1400-USOC-STATUS-OK]	√
	Are actions that can be executed in parallel modelled as parallel flows?		√
	Are actions that can be executed in sequential flow modelled as sequential flows?		√

Method	Criteria	Example	Present
	If two or more states that are in parallel flows succeed to a sequential executing successive states, the preceding states in parallel flows must transition to a merge synchronisation bar and then from this bar, a transition to the sequential successive state. Is this modelled in all of these cases?		√
	If a sequentially-executing state transitions to successive states that can execute in parallel, the preceding, sequentially-executing states must transition to a synchronisation fork bar. From this bar, a transition from the bar to parallel-executing successive states which are modelled as parallel flows. Is this modelled in all of these cases?		V
	Does each diagram have an appropriate start and end state?		√
	Each state, other than start or end state, represents a WSL codeline. Does each state, other than the start and end states, have at least one transition to and at least one transition from?		√

Appendix I: Publications by Candidate

During the duration of my research, I was able to present several publications regarding my work to the academic community. These are the following:

1) Millham, Richard "An Investigation: Reengineering Sequential Procedure-Driven Software into Object-Oriented Event-Driven Software through UML Diagrams", *Proceedings of the International Computer Software and Applications Conference,* Oxford, 2002.
2) Millham, Richard, Hongji Yang, Martin Ward "Determining Granularity of Independent Tasks for Reengineering a Legacy System into an OO System", *Proceedings of the International Computer Software and Applications Conference*, Dallas, Texas, 2003.
3) Millham, Richard, Hongji Yang "TAGDUR: A Tool for Producing UML Diagrams Through Reengineering of Legacy Systems" , *Proceedings of the IASTED Conference on Software Engineering*, Marina del Ray, USA, 2003.
4) Pu, JianJun, Richard Millham, Hongji Yang "Acquiring Domain Knowledge in Systematising the Process of Reverse Engineering Legacy Code into UML", *Proceedings of the IASTED Conference on Software Engineering*, Marina del Ray, USA, 2003
5) Millham, Richard, JianJun Pu, Hongji Yang "TAGDUR: A Tool for Producing UML Sequence, Deployment, and Component Diagrams Through Reengineering of Legacy Systems", *Proceedings of the IASTED Conference on Software Engineering*, Innsbruck, Austria, 2005
6) Pu, Jianjun, Hongji Yang, Steve McRobb, Richard Millham, "Visualising COBOL Legacy Systems with UML: An Experimental Report", in Advances in UML-Based Software Engineering, IDEA Group, Hershey, PA, USA, 2005