Incorporating Quality Requirements in Software Migration Process*

Ying Zou
Dept. of Electrical & Computer Engineering
Queen's University
Kingston, ON, K7L 3N6, Canada
zouy@post.queensu.ca

Abstract

The reengineering of legacy software systems to modern Object Oriented platforms has received significant attention over the past few years. However, most often the reengineering process ignores to take into account specific non-functional requirements, such as maintainability, *for the target system. In this paper, we propose a quality driven software migration framework that aims to identify and extract an object model from a procedural system and to generate quality migrant object oriented code which satisfies non-functional requirements. Such a reengineering framework is composed of quality models to denote desired quality characteristics for the target migrant systems, transformation rules and, an incremental and iterative quality-driven transformation process that is based on a state transition system. The process aims to identify a sequence of software transformations that generate a target system with desired quality characteristics. Moreover, an evaluation technique is presented to verify and validate the achievement of quality requirements in the obtained migrated system. The result obtained from case studies demonstrates the effectiveness and usefulness of the proposed quality driven migration technique.*

1. Introduction

Non-functional requirements, such as reusability, maintainability, performance, portability and, security define system properties, constraints and software qualities of the system being developed. These non-functional requirements play an important role in the software development process. Their achievement in the source code ensures the success of software products. However, as software systems evolve, system properties and qualities are constantly deteriorating and deviating from their original specifications due to prolonged maintenance activities and technology updates. To leverage business values entailed in such systems, a promising solution is to migrate selected parts of these systems to modern platforms and designs. In this context, there are always questions posed and concerns raised regarding whether the newly migrated system will posses as good or better characteristics as the original system. For example, whether the new system can run as fast as the old system or whether the new system is as maintainable as the old one. To address such questions, software migration activities should not occur in a vacuum. It is important to incorporate non-functional requirements in the migration process.

In this paper, we present a systematic and quantitative approach that directs the migration process to produce quality software in the target system. The migrated system conforms to specific quality requirements, such as better reusability and higher maintainability. In this quality driven migration process, we are concerned with three major issues: *1)* modeling the quality goals required in the target system, *2)* operationalizing these quality goals in the migration process, and *3)* validating these quality goals in the target system.

To address the first issue, the desired quality goals are elicited based on domain knowledge, customer interviews and documentation. It is critical to effectively model and refine high level quality requirements in a way that facilitates the validation of whether the quality goals have been achieved in the target system. To address the second issue, a set of transformations is identified to detect code features and improve the qualities of the target system. For example, source code features such as *gotos* can be problem prone artifacts that violate the target goals. Each transformation is selected and applied according to its potential impact on the desired qualities in the target migrant system. Finally, to assess whether the desired goals have been achieved in the target system, a set of software metrics and source code features are evaluated.

To achieve these objectives, we devise a tractable methodology whereby source code transformations can be associated with specific quality improvements and be applied for the migration of procedural systems to object oriented platforms.

*This is a summary of the author's Ph.D. thesis funded by IBM Centers for Advance Studies and CSER.

The rest of the paper is organized as follows. Section 2 investigates related work in literature. Section 3 presents an overview of a quality driven migration framework. Section 4 describes software quality modeling process. Section 5 describes transformation rules that migrate procedural code into object oriented platforms. Section 6 provides an approach to integrate quality goals in the migration process. Section 7 discusses the experiments. Finally, section 8 concludes the paper.

2. Related Work

2.1 Migration Procedural Code to Object Oriented Platform

In the relevant literature, several methods for identifying an object-oriented model of a legacy system have been defined.

In [1] an approach to identify object-oriented model from RPG programs is presented. The class identification is centered around persistent data stores, while related chunks of code of the legacy system become candidate methods. In [2], a semi-automatic tool to migrate PL/IX programs into C++ is presented. The abstract syntax tree (AST) is used to analyze the original source code. Consequently, tokens in AST are transformed into C++. Through a similar approach, an automatic Fortran to C/C++ converter is provided in [3], a Pascal to C++ converter in [4], and methodologies to directly identify C++ classes from procedural code written in C [5]. Moreover, in [5], an evidence model is presented to help the user to select the best migrated object model. This evidence model includes state change information, return types, and data flow patterns.

In [6, 7], concept analysis is proposed to identify modules in C code. It is based on lattices that are used to reveal similarities among a set of objects and their related attributes. However, the size of lattices is considerable large, even for a small scale software system.

In [8], a C to Java migration tool, called Ephedra, is presented. The tool reads C/C++ code in binary format and produces Java source code. It can handle most of C/C++ structures and convert them into Java code automatically, except for goto statements, type casts, unions and templates conversion.

In a nutshell, existing migration methods aim to identify Abstract Data Types (ADT) and extract candidate classes and methods from procedural code. However, there is no comprehensive framework for ensuring that the migrated system posses certain quality characteristics. In this paper we aim to provide a generic reengineering approach that monitors and evaluates the software quality of the system being reengineered at each stage of the migration process to ensure that such quality goals are met in the target system.

2.2 Software Quality Specification Framework

Software quality is defined by a set of features and characteristics of a software product that relate to external attributes, such as performance, and internal attributes such as, the complexity of data structures. The external attributes, mainly qualify the operational environment of a system. The internal attributes relate to source code features and can be measured by a collection of appropriate metrics. External and internal system attributes are cognitively relevant and interdependent. For example, external attributes such as maintainability, depend on internal attributes such as high cohesion and low coupling.

Software quality are also referred to as soft goals, because the degree of the conformance to specific non-functional requirements can be positively or negatively specified in relative terms instead of absolute terms. A soft goal interdependency graph [9] can be used to model interdependencies between quality attributes. Nodes are used to capture informal concepts, and represent goals to be satisfied in order to achieve a desired quality or property. Edges represent dependencies as to how these goals can be satisfied. The goals (i.e., nodes) in the graph require their sub-goals be satisfied only partially for the parent goal to succeed. Soft-goals may depend on sub-goals according to "AND/OR" relations. An "AND" dependency means that all sub-goals need be satisfied to a certain degree for the parent goal to be satisfied. An "OR" dependency means that in order for the parent goal to be achieved any of the sub-goals must be achieved first. A typical Soft-Goal Interdependency Graph for high modularity is illustrated in Figure 2. A process-oriented framework to represent non-functional requirements as soft goals is proposed [9]. The framework consists of five major components: a set of goals for representing nonfunctional requirements that positively or negatively contribute to other goals; a set of link types for relating goals to other languages. a collection of correlation rules for deducing interdependencies between goals; and finally, a labeling procedure which gives a quantitative method to determine any given nonfunctional requirement is being addressed by a set of design decisions.

In [10], a quality-driven software re-engineering framework is proposed. The framework aims to associate specific soft goals, such as performance and maintainability with transformations and guide software reengineering process. This work focuses on providing a catalog of transformations to refactor object oriented code to produce better object oriented designs using design patterns. We provide an approach that focuses on the process of software migration of procedural code to object oriented platforms. We aim to refine high level quality goals such as maintainability and reusability into low level source code features in the target object oriented systems. Moreover, we devise a quantitative method to

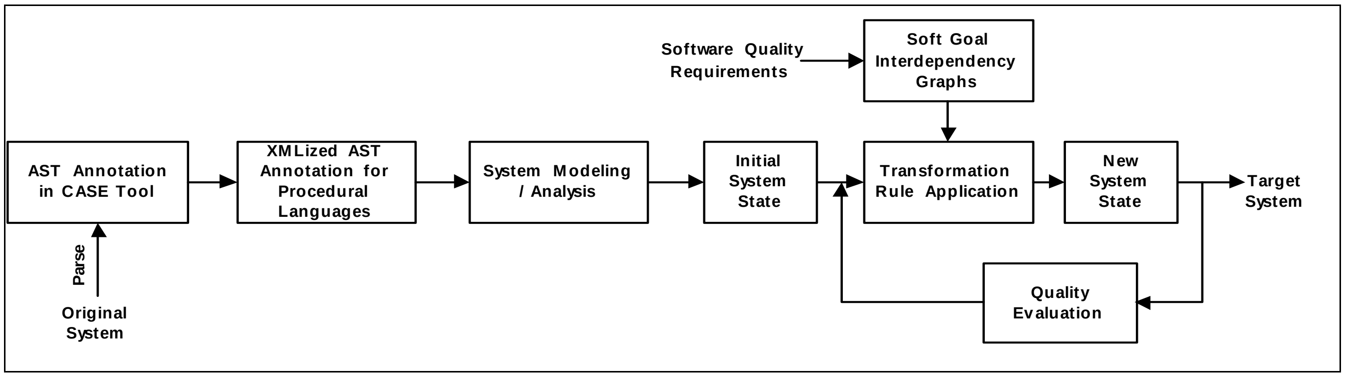

Software Quality Requirements
Soft Goal Interdependency Graphs
AST Annotation in CASE Tool
XMLized AST Annotation for Procedural Languages
System Modeling / Analysis
Initial System State
Transformation Rule Application
New System State
Target System
Parse
Original System
Quality Evaluation

Figure 1: Quality Driven Software Migration Framework

assess the likelihood of each transformation to achieve desired quality goals. Finally, an optimal transformation path can be determined to generate target object oriented systems with the highest quality.

3. Quality Driven Software Migration Framework

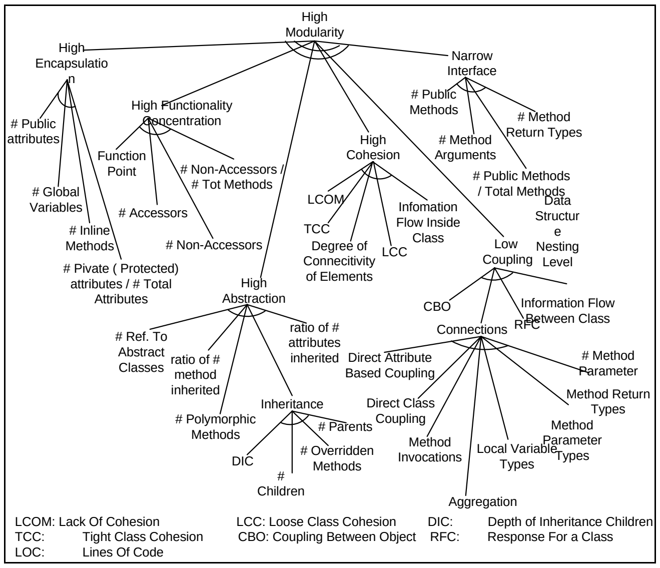
Software migration is a process of examining and altering a subject system to reconstruct it in a new form [11]. The migration activities we are interested in, aim to transform a subject system from its original procedural language implementation to an object oriented design without altering its external behaviors. Moreover, we consider target software quality goals as an integral part of the migration process. Such a quality driven migration process is illustrated in Figure 1.

Initially, the original legacy system is parsed and represented in terms of Abstract Syntax Trees (ASTs) or entity-relationship models. The software entities represent source code features of interest, such as data types, formal and actual parameters, global variables, and functions. The relations denote interactions between software entities, such as function calls, global variable usage, or data type references. The central issue is to represent the procedural code of the system being analyzed in a generic representation. High-level procedural languages, such as C, Cobol, Fortran and Pascal, all comply with different specific domain models that denote the structures and to a certain extent the semantics of each language. However, a variety of procedural languages have semantic equivalent language constructs. We introduce the concept of a unified domain model for a variety of procedural languages such as C, Pascal, COBOL, and Fortran. This unified model is represented in XML and denotes common language features such as routines, subroutines, functions, procedures, types, statements, variables and declarations, just to name a few. In this way, for different procedural language domains, we can provide a set of generic transformations that migrate procedural systems in different procedural languages into functionally similar object oriented systems.

The system modeling / analysis phase aims to evaluate the original procedural system, detect error prone areas, and identify transformation rules for migration tasks. For example, the existence of global variables violates the concept of encapsulation. In this case, error prone areas must be captured and removed in order to conform to target quality goals for the new system.

Furthermore, we consider a migration process as a state transition system, denoted by a sequence of transformations that alter a system being migrated. Specially, at each step of the migration process, a transformation is applied to yield an intermediate system. A transformation detects the error prone features, and generates the desired migrant artifacts. Moreover, we adopt soft goal interdependency graphs to associate high level quality goals with specific source code features. In this case, each transformation is selected to alter source code features in the original system, and assessed according to its potential impact on the desired qualities for the target system. Consequently, the resulted system is evaluated against the desired goals. Finally, the process terminates either when the highest achievable measured level of quality for the target system has been reached or no transformations can be further applied.

In short, the proposed migration framework consists of four models. First, a software quality model allows for the association of specific system qualities with source code features, and design decisions. Second, a software representation model allows for the representation of various procedural languages in a generic and uniformed format. Third, a transformation representation model allows for the denotation of software transformation rules in a generic form. Finally, transformation process model allows for the selection and application of transformation rules, so that the target system state has the highest likelihood of conformance to the desired qualities. In the following sections, we discuss these models and techniques in more detail.


High Modularity
High Encapsulatio n
# Public attributes
# Global Variables
# Inline Methods
# Pivate (Protected) attributes / # Total Attributes
High Functionality Concentration
Function Point
# Accessors
# Non-Accessors / # Tot Methods
# Non-Accessors
High Abstraction
# Ref. To Abstract Classes
ratio of # method inherited
# Polymorphic Methods
ratio of # attributes inherited
Inheritance
DIC
# Children
# Overridden Methods
# Parents
High Cohesion
LCOM
TCC
Degree of Connecitivity of Elements
LCC
Infomation Flow Inside Class
Narrow Interface
# Public Methods
# Method Arguments
# Public Methods / Total Methods
# Method Return Types
Low Coupling
Data Structur e Nesting Level
CBO
Connections
RFC
Information Flow Between Class
Direct Attribute Based Coupling
Direct Class Coupling
Method Invocations
Aggregation
Local Variable Types
Method Parameter Types
Method Return Types
# Method Parameter
LCOM: Lack Of Cohesion
TCC: Tight Class Cohesion
LOC: Lines Of Code
LCC: Loose Class Cohesion
CBO: Coupling Between Object
DIC: Depth of Inheritance Children
RFC: Response For a Class
Figure 2: Soft-goal Interdependency Graph of High Modularity

4. Software Quality Modeling

To guide the migration process to meet quality objectives, we provide a software quality modeling process that elicits quality goals and models quality in a measurable level, and evaluates the achievement of desired qualities in the new resultant systems. The proposed quality modeling process includes the following steps.

1) *Assess quality status:* a software quality assessment is an attempt to analyze and describe a software system's quality from different perspectives: its characteristics, strengths and weaknesses [12]. Especially, for the migration process, we aim to preserve the system's strengths and enhance its weakness. Software metrics are widely adopted for quality assessments. Moreover, the result of the assessments facilitates the prediction of migration efforts.
2) *Identify critical quality bottlenecks:* the critical quality bottlenecks refer to error prone areas, such as design flaws, which are important to operate, maintain or reuse target systems. The critical bottlenecks are subjective, and specific to the target domain. In particular, different bottlenecks may lead to conflicts towards achieving desired qualities goals. In this context, the key bottlenecks can be rated based on the domain knowledge.
3) *Establish quality objectives:* quality objectives are generally sufficient for addressing serious quality problems, such as bottlenecks, at the architectural level, design level or code level. The selected objectives specify measurable criteria for evaluating quality characteristics in target software systems. Typically, quality objectives are established in a top down manner, which involves identifying a set of high-level quality goals, such as functionality, maintainability, as specified in ISO 9126, subdividing and refining them into more specific attributes (e.g., design decisions, software code) at lower levels. The lowest level attributes can be directly measurable using software metrics. Moreover, quality objectives are selected according to critical quality bottlenecks and target application scenarios. For example, object oriented designs offer properties such as encapsulation, inheritance, and polymorphism. These properties make software systems easier to reuse and maintain. Therefore, in the scenario of migrating procedural code into object oriented platforms, we aim to achieve two quality objectives namely, reusability and maintainability.
4) *Construct quality models:* quality models provide a representation that depicts the quality refinement process for the selected objectives. Typically, one quality model is used to describe one quality objective. We adopt soft goal interdependency graphs to build software quality models that link a rationale on whether a transformation can achieve a specific non-functional requirement by associating dependencies between non-functional requirements and internal source code attributes that are altered by a given transformation. An example software quality model as illustrated in Figure 2 demonstrates that source code features in the leaf nodes contribute to the high modularity goal. Furthermore, the soft goal model assists in evaluating the likelihood that a transformation satisfies a specific desired soft goal in the migrated system. In general, soft goals are achieved in a top-down manner satisfying the appropriate sub-goals on each step of the process. In this context, changes to code features by transformations in the lower levels of the soft goal hierarchy can be traced up to reflect changes in the root of soft goal interdependency graphs.
5) *Quality measurement:* For the software quality modeled in a soft-goal interdependency graph, a set of metrics are selected to compute the corresponding source code features, appearing as leaves in the soft-goal interdependency graph. The metric results of the leaf nodes indirectly reflect the satisfaction of their direct or indirect parent nodes in the soft-goal interdependency graph. Furthermore, each re-engineering transformation is associated with a collection of features it affects and consequently with the magnitude of change in the corresponding quality being affected. In the context of our research, we are interested in examining a number of product metrics and features that are related to reusability and maintainability. For example, we aim to achieve high cohesion, and low coupling. These quality goals are considered as sub-goals, and consequently achieve higher-level goals such as reusability and

Metric	Name	Definition	Cite
IFIC	Information Flow Inside Class	IFIC computes the sum of each method in a class that the number of invocations of other methods of the same class, weighted by the number of parameters of the invoked method. A high value is desired.	[18]
TCC	Tight Class Cohesion	TCC computes the percentage of pairs of public methods of the class which share the same attributes in their own class, i.e., pairs of methods. A high value is preferred.	[19]
LCOM	Lack of Cohesion among Methods	Consider a set of method M_i (i = 1, …, m) accessing a set of attribute A_j (j=1, …, a). Let $\mu(_{Aj})$ be the number of methods which refer to attribute A_j. Then $LCOM = \frac{1}{a}(\sum_{j=1}^{a} \mu(A_j) - m/(1-m)$ .A low value is desired.	[20]
Coh	Cohesion Measurement	Consider a set of method M_i (i=1,…, m) accessing a set of attributes A_j (j=1,…, a). Let $\mu(_{Aj})$ be the number of methods which refer to attribute A_j. Then $Coh = (\sum_{j=1}^{a} \mu(A_j))/(m \cdot a)$. A high value is desired. Coh is another form of LCOM.	[20]
NPTIC	Number of Parameter Types Inside Class	NPTIC computes the total number of method parameter types referring to their own class type. A high value is desired.	[5]
NMRTIC	Number of Method Return Types Inside Class	NMRTSIC computes the total number of method return types referred to their own class type. A high value is desired.	[5]
NLVTIC	Number of Local Variable Types Inside Class	NLVTSIC computes the total number of local variable types referring to their own class type. A high value is desired.	[5]
NMUAIC	Number of Methods Updating Attributes Inside Class	NMUAIC computes the total of methods that directly update attributes in their own class.	[5]

Table 1: Selected Metrics & Features for Measuring Cohesion

maintainability, as depicted in [13]. Tables 1 and 2 list selected metrics and source code features which guide the application of re-engineering transformations.

5. Transformation Representation Model

For the source code to source code reengineering, a series of transformations can be applied to either restructure a system and generate better or optimized code in the same language domain, or convert it into a new language domain. In both approaches, a transformation takes a source code construct in the original procedural system as input, and yields migrated software artifacts as output. In

Metric	Name	Definition	Cite
CBO	Coupling Between Object	a class is coupled to another, if methods of one class use methods or attributes of the other, or vice versa. CBO computes the number other classes to which a class is coupled. This can include the inheritance based coupling. A low value is preferred.	[21]
DCC	Direct Class Coupling	DCC counts the different number of classes that a class is directly related to. The relation between classes includes the data attribute declaration and message passing in methods. A low value is preferred.	[20]
IFBC	Information Flow Based Coupling	IFBC computes the number of external method invocations in a class, weighted by the number of parameters of the invoked methods. A low value is desired.	[18]
NPTOC	Number of Parameter Types Outside Class	NPTOC computes the total number of method parameter types referring to other classes. A low value is desired.	[5]
NMRTOC	Number of Method Return Types Outside Class	NMRTOC computes the total number of method return types referred to other classes. A low value is desired.	[5]
NLVTOC	Number of Local Variable Types Outside Class	NLVTOC computes the total number of local variable types referring to other classes. A low value is desired.	[5]
NMUAOC	Number of Methods Updating Attributes Outside Class	NMUAOC computes the total of methods that directly update attributes in other class. A low value is desired.	[5]

Table 2: Selected Metrics & Features for Measuring Coupling

our research, we aim to apply unified transformations on various procedural language domains, and generate object oriented code. Within this context, we create a generic code representation for procedural languages. Therefore, transformations and source code features can be described abstractly and applicable in our generalized procedural code representation. We define a transformation representation model to specify transformation rules, source code features, and their linkages.

Motivated by the relation between classes and objects in the object oriented design, we model procedural source code features and migrated object oriented code features as class templates and transformation rules as association classes. Transformation rules link source code features in the procedural language domain with the source code features in an object oriented language domain. The class templates and association classes can be instantiated as objects based on a concrete source code context upon which they are applied. In this respect, a transformation is an instance of a specific transformation rule. Moreover, to efficiently specify constraints to apply transformation rules in procedural code features, we associate pre- and post-conditions with each transformation rule. Pre-conditions specify the rights of a transformation that offers the operation. A pre-condition must be true before a transformation is applied. Similarly, the obligations are specified by post-conditions. A post-condition must be true when a transformation has just ended its operation. In the migration process, pre-conditions are the procedural source code features that a transformation can operate on and convert them into object oriented structures, as specified in post-conditions. We adapted Unified Modeling Language (UML) to model source code features and transformation rules as class templates and association classes. Furthermore, OCL (Object Constraint Language) is chosen to specify well-formed pre-conditions and post-conditions for the transformation rules.

Moreover, we identified and formally specified a set of transformation rules in two categories, namely, class creation and object model refinement.

5.1. Object Identification

The rules on the class creation category define the criteria for the production of object models from the procedural code. In the search of an object model to extract from procedural source code, we aim at achieving high encapsulation, high cohesion within a class, and low coupling between classes. In an object-oriented system, software is structured around data rather than around services of systems. Therefore, the object model extraction process is divided into three steps: class identification, private data member identification and method attachment.

The first step towards the migration of a procedural system to an object-oriented system is the selection of possible object classes. The **object identification techniques** focus on two areas: a) the analysis of global variables and their data types, and b) the analysis of complex data types in formal parameter lists. Analysis of global variables and their corresponding data types focuses on the identification of variables that are globally visible within a module. For each variable, its corresponding type is extracted from the Abstract Syntax Tree, and a candidate object class is generated. Data type analysis focuses on type definitions that are accessible via libraries. Examples include typedef C constructs. Data types that are used in formal parameter lists become also primary class candidates. The union of data types that are identified by the global variable analysis and data type analysis forms the initial pool of candidate classes. Furthermore, based on the multiple usage of a set of related data types in formal parameter lists, this set of data types can be encapsulated into one aggregate class to minimize the interface of the methods, which take the set of data types as parameters instead.

In the second step, the data fields in the user defined aggregate types and the global variables are converted into data attributes of the corresponding class candidates.

Finally, the procedural functions are assigned into class candidates as methods. To achieve higher encapsulation, it is desirable that data members in class candidates can only be directly used or modified by the methods in the same class candidate, instead of by methods in other class candidates. Therefore, procedural functions are attached to class candidates based on the usage of data types in the formal parameter lists, return types, and variable usage in the function body. For this task, only aggregate data types in the parameter list and return value are considered. Simple types such as int, char, and float are ignored.

5.2. Object Model Refinement

Inheritance and polymorphism are some of the most important features of an object-oriented design. To achieve a good object-oriented design from a procedural legacy system, we have identified a number of transformation rules that can be used to establish associations between abstract data types and refine the identified object model.

To identify inheritance relations, we examine procedural source features that indicate a similar concept as inheritance. For example, we detect cast operations between abstract data types. In the case of cast operations, the compiler will automatically change one type of data into another when appropriate. Casting allows to make this type conversion explicit, or to force it when it wouldn't normally happen. Implicit cast operation between two data types suggests that these data types share common data fields or are interchangeable. As a consequence, the inheritance is established by taking the data type with more specified data fields as a child class candidate and data type with more general information as a parent class candidate. Moreover, other code features indicate inheritance including a *struct* type defined inside other *struct* types and date field clones among *struct* types.

To discover polymorphism relations, we examine procedural code features that imply the same properties as polymorphism, including function pointers, switch statements, conditional statements, and generic pointer parameters. For example, function pointers in procedural code are pointers (i.e. variables) which point the address of a function. Function pointers allow a procedural function to be passed as a parameter. The actual function to be invoked is determined at run-time based on the type of the variable. In this respect, function pointers share a similar mechanism as polymorphism that allows a variable to take different types dependent on the context at execution time. In the object oriented migration process, each possible function pointer reference can become a class and their corresponding source code can become a polymorphic method.

6. Quality Driven Migration Process

In our work, we aim to construct a quality driven migration process that takes into account the need for the migrated system to conform to specific constraints and yield a migrated system that has a high likelihood of obtaining desired quality characteristics. In this section, we discuss a transformation process model that allows for the selection and application of transformation rules. Moreover, we provide a quantitative approach to indicate transformation impact on specific quality objectives.

6.1. Transformation Process Model

Conceptually, a migration process is considered as a sequence of transformations, t_{01}, t_{02}, …, t_{ij}, $t_{i,j+1}$, …, t_{kn},, that alter the system's states, s_0, s_1, …, s_i, s_{i+1}, …, s_n, as shown in Figure 3. In this sense, we represent the migration process as a labeled transition system, and refer to it as a transformation process model. A labeled transition system [14] is denoted as $(S, T, \{\xrightarrow{t} \mid t \in T\})$, which consists of a set of states S, a set of transitions T, and a transition relation $\xrightarrow{t} \subseteq S \times T \times S$. The initial state s_0, corresponds to the original system and the final states corresponds to a set of target migrant systems. I refers to the initial states, and F denotes a set of final states. The states represent the evolution of the subject system from the initial system to the final target system. The transitions are carried out by a comprehensive set of possible transformations. For the process in the migration of procedural code to object oriented platforms, the transition relation is defined by transformation rules that examine the source code and extract the candidate classes and their associations from the procedural code (see Section 5).

To incorporate quality control in the migration process, each state can be specified by a set of source code features related to one or more target qualities (i.e., performance, maintainability, and reusability). Furthermore, each state is denoted by a set of feature vectors, v_i =<(f_1,a_1), (f_2,a_2), …, (f_k,a_k), …, (f_2,a_m)>. a_i is a metric value that quantifies the source code feature f_i that appears as a leaf node in the soft-goal interdependency graphs and relates to the desired quality (i.e. maintainability and reusability) of the target migrated system. Each feature vector v_i is used to evaluate and relates to one goal, or one sub goal. Therefore, multiple feature vectors can be associated with a state when more than one goal or sub-goal is considered. Furthermore, a feature vector provides a quality signature for each state. In the quality driven migration process, we distinguish two states in terms of

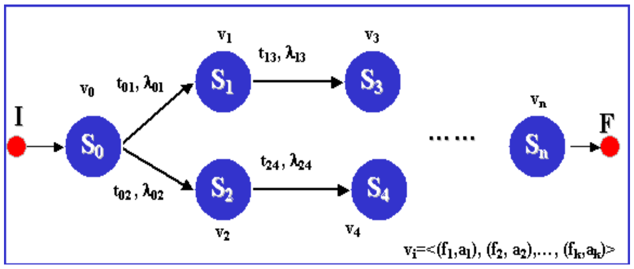


Figure 3: Quality Driven Software Migration Framework

their qualities represented by feature vectors. As illustrated in Figure 3, each transformation, t_{ij} , is denoted by a likelihood score λ_{ij} that is used to denote the effects of a transformation towards achieving a specific quality. Moreover, the likelihood value facilitates the design of algorithms to select a specific transformation among alternatives towards the desired goals. It is also worth to mention that the likelihood is not equivalent to a probability that considers all the possible occurrences of events. To this end, the likelihood is calculated based on the considered quality attributes and available transformations. At this point, it is not proven that likelihood values follow the laws of the probability theory. More specifically, a transformation may cause the value a_k in a feature vector v_k to increase, decrease, or stay unchanged. As a consequence, the change is quantified by a delta on the corresponding feature vectors in two consecutive states. The more features that are altered positively by a given transformation, the higher the quality score and the likelihood that the transformation can contribute towards the desired quality objectives.

We proposed two formulas to calculate likelihood. Formula (1) is proposed to evaluate the measure of $\lambda_{(G)ij}$ which denotes the estimated level that the transformation t_{ij} improves the quality characteristics of a system with respect to the quality goal G. We calculate the deltas of the source code changes caused by a transformation in two consecutive states. Moreover, the formula (2) is used when a goal is a composition of a set of sub-goals.

$$\lambda_{(G)ij} = e^{\frac{\sum \text{Positvely Affected Features} - \sum \text{Negatively Affected Features}}{\sum \text{Features}}} \qquad (1)$$

$$\lambda_{(G)ij} = e^{\sum_{k=1}^{m} c_k \lambda_{(k)ij}} \qquad (2)$$

where m is the total number of the goals, c_k is the coefficient for each goal(k) and $\lambda_{(k)ij}$, is the likelihood for the transformation t_{ij} to achieve goal(k).

6.2. Transformation Path Selection

A transformation is triggered and applied once source code features in a state meet preconditions of the transformation. Each transformation manipulates a set of source code features, migrates such features into the corresponding object oriented constructs, and incrementally updates the object model to enclosure the newly identified object oriented constructs. The transformation process ends when there are no source code features that can invoke transformations or an object oriented system is obtained. In this case, a full transformation path can be formed by the composition of transformations from original state to the final states. An example transformation chain and the associated likelihood values are depicted in Figure 4. Moreover, the effect of transformations on quality objectives is measured throughout the whole migration process. As illustrated in the diagram, every transformation is labeled with a likelihood score.

As depicted in the example, there are multiple transformation paths originating from the initial state to the final states. A likelihood of a transformation path can be calculated by multiplying the likelihood of each transformation along the path. An optimal transformation path has the highest likelihood towards achieving the desired qualities. Finally, the migrant system is chosen from the optimal path.

We adapted Viterbi algorithm [15] to generate all possible transformation paths and identify the optimal sequence of transformations. However, this may not be suitable for large systems where a large number of states are generated. In addition, a large system can be divided into a set of smaller clusters. The migration process is conducted incrementally cluster by cluster [16]. To this end, the Viterbi algorithm is applied in each cluster with the reduced search space. The selection of the optimal path can be also based on traversing the model and applying a simulated annealing search [17]. In this case, not so promising transformations can be also considered with the expectation that may later yield a state that can produce an optimal system.

7. Quality Evaluation Techniques

In the context of quality evaluation, multiple alterative systems resulting from different migration paths are generated. A set of selected object oriented metrics can be utilized to evaluate the quality of each system. In particular, we are interested in validating that the optimal transformation path can generate object oriented system with the highest quality. To achieve this objective, all transformation paths are ordered in a sequence based on the values of likelihood. Every path is assigned with a number in the range of 0 and 1. Such number shows a relative position of a path in the sequence. For example, the path labeled with 1 refers to the optimal path with the highest likelihood. The path labeled with 0 refers to the worst path with the lowest likelihood. Similarly, the path of 0.5 means the medium path whose likelihood is ranked in the middle of the highest likelihood and the lowest one.

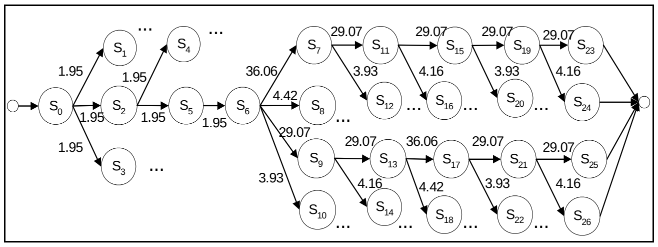


Figure 4: An Example Transformation Chain

Systems	Apache	Bash	Clips
Lines of Code	37,033	27,521	34,301
Num. of Files	42	39	40
Num. of Functions	709	998	736
Num. of Aggr. Types	184	79	151
Num. of Global Vars	103	227	186

Table 3: Characteristics of the Procedural Systems

The path of 0.75 has the likelihood ranked in the middle of the optimal path and the medium path. Likewise, the path of 0.25 has the likelihood ranked in the middle of the medium path and the worst path. In such a way, we can generate five target systems from the paths of 0, 0.25, 0.5, 0.75 and 1. Every target system has the same number of classes. Every class in one target system can be found in the other four systems with the same class name and data members. However, the method members might be different because each system chooses alternative classes to assign the conflicting methods based on following different transformations. To this end, we create a common ground, (i.e. the similar class structure) to compare metric results from alterative systems. The differences in method assignments affect the cohesion, coupling, and encapsulation quality attributes of migrated systems.

8. Experiment

To investigate the effectiveness of the quality driven migration framework, a prototype software toolkit is developed. This toolkit developed a system segmentation algorithm to break a large system into a set of smaller working areas. Therefore, the quality driven migration process is applied iteratively in each cluster and allows the procedural systems to be migrated into object oriented platforms incrementally. The case studies are performed on three medium size open source procedural systems from different domains: 1) The Apache Web Sever 2) The BASH Unix Shell and 3) The Clips Expert System Builder. Table 3 presents the source code related characteristics of the systems.

8.1. Experiment Steps

For our experiment, the migration process is performed in the following order:

- Identification of all possible class candidates: these

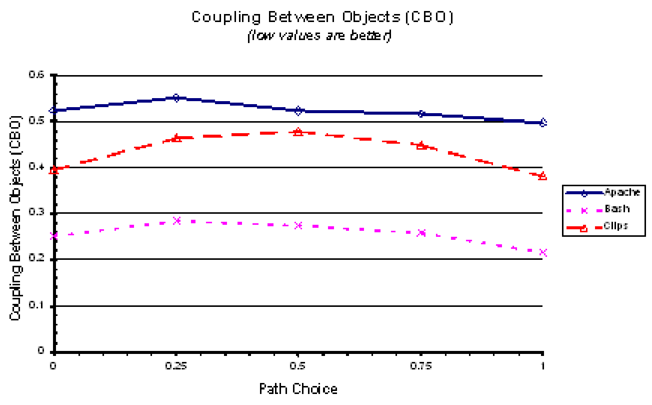


Figure 5: Coupling Evaluation for Migrant Systems

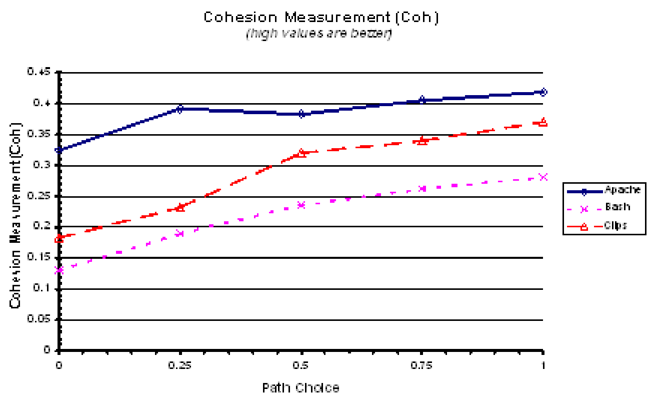


Figure 6: Cohesion Evaluation for Migrant systems

class candidates form the initial state. In this case, one method might be assigned into more than one class candidates. This leads to the generation of multiple states of which each state is produced by assigning a conflicting method into one alternative class. The order of resolving methods in conflict is determined by the function call relations in the methods. The most called methods by other methods in conflict are resolved first. Therefore, an ordered sequence of methods in conflict is produced by the reengineering toolkit.

- Generation of all possible transformation paths: Each transformation path consists of a sequence of transformations to migrate a procedural code into the object oriented design. Each path generates an alternative object model. Every transformation is labeled with a quality likelihood.
- Quality evaluation of design alternatives: Each output object model from one transformation path is evaluated in terms of coupling and cohesion qualities.
- Code generation from the target object model: Object oriented code is generated for the optimal migrated system.

8.2. Likelihood Computation

We select a set of source code features to compute the likelihood of each transformation to achieve high cohesion and low coupling. These source code features are not co-related. Specifically, to compute the likelihood to achieve high cohesion we consider the features, including NPTIC (Number of Parameter Types Inside Class), NMRTIC (Number of Method Return Types Inside Class), NLVTIC (Number of Local Variable Types Inside Class), NMUAIC (Number of Methods Updating Attributes Inside Class), and IFIC (Information Flow Inside Class), as listed in Table 1. Similarly, to compute the transformation likelihood to achieve low coupling, the source code features include NLVTCO (Number of Parameter Types Outside Class), NMRTOC (Number of Method Return Types Outside Class), NLVTOC (Number of Local Variable Types Outside Class) NMUAOC (Number of Methods Updating Attributes Outside Class), and IFBC (Information Flow Based Coupling) as listed in Table 2. Based on formula 1, the likelihood values for cohesion and coupling are calculated individually based on the deltas of these source code features in two consecutive states. Furthermore, the cumulative result of both goals is combined by the formula 2.

Systems	Apache	Bash	Clips
Num. of Classes from Global Variables	103	227	186
Num. of Classes from Struct Types	184	79	151
Num. of Classes from Function Pointers	4	4	3

Table 4: Characteristics of Migrated Systems

8.3. Quality Evaluation of the Migrant Systems

For the purpose of evaluation, we measure the coupling and cohesion properties in classes. We choose a different set of metrics from the one we used to compute the transformation likelihood. In this respect, the coupling between classes is measured by three metrics, namely, CBO (Coupling Between Objects), DCC(Direct Class Coupling) and IFBC (Information Flow Between Classes) as listed in table 2. Moreover, the cohesion inside a class is measured by metrics, including TCC(Tight Class Cohesion), Coh(Cohesion Measurement) and IFIC (Information Flow Inside Class) as listed in Table 1.

To reflect the overall result of the entire system with respect to one specific metric, we calculate the average of the metric result. Some sample results for coupling and cohesion evaluation are illustrated in Figure 5 and Figure 6. The X-axis shows the choices of paths. The Y-axis represents metric values corresponding to the systems generated from the five selected paths. Figure 5 illustrates the measurement for coupling between objects. In this case, the lower value is better. As shown in the result, the optimal path labeled as 1 has the lowest value

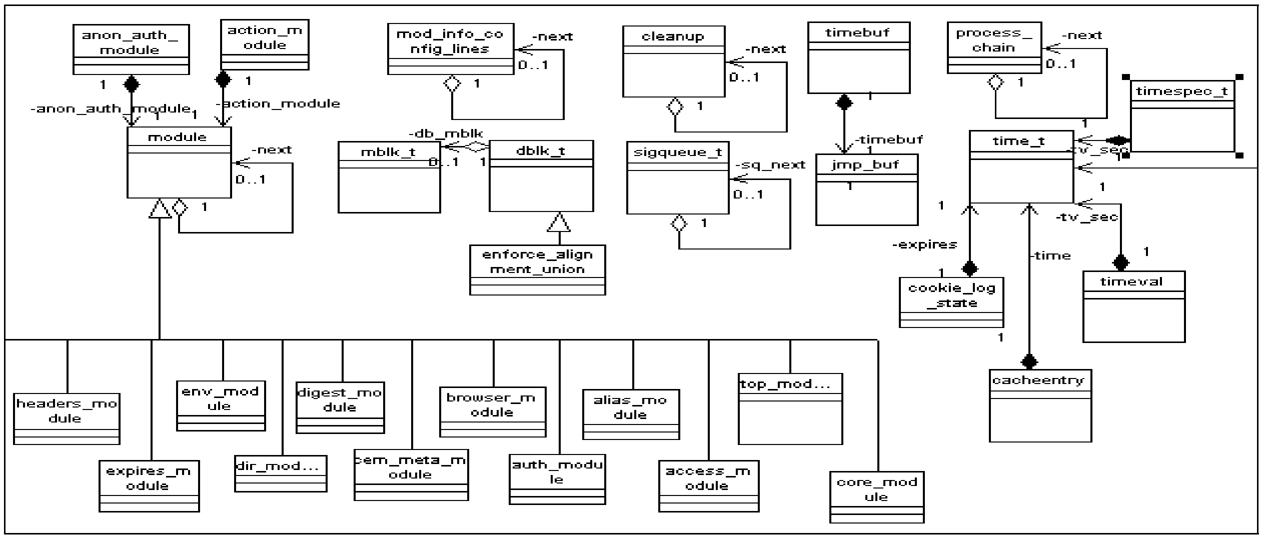


Figure 7:Class Structure of Migrated Apache Web Server

among the alterative systems (i.e., the systems labeled with 0, 0.25, 0.5, and 0.75). The same result holds in the three examined systems. Figure 6 depicts the measurement for cohesion. In this case, the higher value of metric results is better. The optimal path has the highest value among the five alterative systems. Moreover, by combining results from all the six metrics, we conclude that the final object oriented system generated from the optimal path has the highest qualities comparing with the object oriented systems generated from the alternative paths.

8.4. Generation of Object Oriented Systems

Once the migration process is completed, the final object oriented code is generated from the optimal transformation path. In this stage, human assistance can be considered to adjust class structure and inheritance by incorporating domain knowledge. Finally, the compliable code is imported into Rational Rose where UML class diagrams can be automatically generated. Table 4 summarizes the characteristics of the classes in all three migrant systems, whereas Figures 7 illustrates the obtained object models of the migrated Apache Web Server.

9. Conclusion

A quality driven migration framework is presented to incorporate quality attributes in the migration process, and to ensure that the target system has the highest desired quality. We believe that future research on validating the achievement of the functional requirements in the target migrant systems is also important. The behaviors of the migrant system should conform to the ones of the original procedural system. We plan to focus on extracting behavior model for the original system, and to trace these behaviors in the migrated systems.

Acknowledgment

I would like to thank my Ph.D. thesis supervisor, Dr. Kostas Kontogiannis for his valuable contribution and support to this work/research.

References

[1] De Lucia, G.A. Di Lucca, A.R. Fasolino, P. Guerra, S. Petruzzelli, "Migrating Legacy Systems toward Object-Oriented Platforms", In the Proceedings of Internatonal Conference on Software Maintenance, 1997.

[2] Kostas Kontogiannis, et. al., "Code Migration Through Transformations: An Experience Report", IBM CASCON 1998.

[3] S.I. Feldman, "A Fortran to C Converter", AT&T Technical Report No. 149, 1993.

[4] "Pascal to C Converter", http://www.garret.ru/~knizhnik/ptoc/Readme.htm.

[5] K. Kontogiannis, P. Patil, "Evidence Driven Object Identification in Procedural Systems'', STEP'99, September 1999, pp. 12-21.

[6] C. Lindig and G. Snelting, "Assessing Modular Structure of Legacy Code Based on Mathematical Concept Analysis", In Proc. Of International Conference on Software Engineering, 1997.

[7] H. A. Sahraoui, W. Melo, H. Lounis, F. Dumont, "Applying Concept Formation Methods To Object Identification In Procedural Code", In Proc. Of 12th Conference on Auotmated Software Engineering, 1997.

[8] Johannes Martin, "Ephedra – A C to Java Migration Environment", http://ovid.tigris.org/Ephedra/.

[9] J. Mylopoulos, L. Chung and B. Nixon, "Representing and Using Nonfunctional Requirement: A Process-Oriented Approach", IEEE Transactions on Software Engineering, Vol 18 No. 6, June 1992.

[10] Ladan Tahvildari, Kostas Kontogiannis, John Mylopoulos, “Requirements-Driven Software Reengineering", *8th IEEE Working Conference on Reverse Engineering*, October 2001.

[11] Elliot J. Chikofsky and James H. Cross II, “Reverse Engineering and Design Recovery: A Taxonomy, IEEE Software, vol. 7, January, 1990.

[12] Thomas E. Volman, “Software Quality Assessment and Standards”, Computer, Vol. 26, No. 6, June 1993.

[13] Ying Zou, Kostas Kontogiannis, “Migration to Object Oriented Platforms: A State Transformation Approach”, Proceedings of IEEE International Conference on Software Maintenance (ICSM), Montreal, Quebec, Canada, October 2002, pp.530-539.

[14] Holgoer Hermanns and Joost-Pieter Katoen. Automated compositional markov chain generation for a plain-old telephone system. Science of Computer Programming, 36(1), 2000.

[15] Paul Alphen and Dick Bergem. Markov models and their application in speech recognition. Technical report, Reader Colloquium Signaalanalyse en Spraak, IPO report 765, 1992.

[16] Ying Zou, Kostas Kontogiannis, "Incremental Transformation of Procedural Systems to Object Oriented Platforms", to appear in the IEEE 27th Annual International Computer Software and Applications Conference (COMPSAC), Dallas, Texas, USA, November 3-6, 2003.

[17] Stuart Russell and Peter Norvig, Artifical Intelligence, A Modern Approach, Prentice Hal, 1995, Englewood Cliffs.

[18] Y.S Lee, B.S. Liang, S.F. Wu, and F.-J. Wang. Measuring the coupling and cohesion of an object-oriented program based on information flow. In the Proceedings of International Conference on Software Quality, Maribor Slovenia, 1995. IEEE Computer Society.

[19] J.M. Bieman and B.-K. Kang. Cohesion and reuse in an object-oriented system. In the Proceedings of ACM Symposium on Software Reusability. ACM, 1995.

[20] Lionel C. Briand, J¨urgen Wst, and Hakim Lounis. Replicated case studies for investigating quality factors in object-oriented designs. Empirical Software Engineering: An International Journal, 6, 2001.

[21] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. IEEE Transactions on Software Engineering, 20(6), June 1994.