Quality Driven Transformation Compositions for Object Oriented Migration

Ying Zou, Kostas Kontogiannis
Dept. of Electrical & Computer Engineering
University of Waterloo
Waterloo, ON, N2L 3G1, Canada
{yzou, kostas}@swen.uwaterloo.ca

Abstract

Reengineering legacy software systems to object oriented platforms has received significant attention over the past few years. In this paper, we propose a goal driven software migration framework that aims to identify and extract a quality object model from a procedural system and to generate quality object oriented code. The framework is composed of analysis tools, transformation rules, and non-functional requirement models for the target migrant system. Specifically, to facilitate the design and development of such goal driven migration framework, source code transformation rules are associated with a degree of belief that they contribute towards enhancing a desired property for the target system. The migration process applies a search algorithm that is guided by the source code analysis to select a transformation sequence that has the highest likelihood of yielding such a target system. The migration of a selected set of gnu AVL libraries to a new object oriented platform is presented as a proof of concept for the proposed technique.

1. Introduction

Object oriented (OO) reengineering focuses on the transformation of procedural software into a functionally similar object-oriented program. In a nutshell, the migration process aims to identify Abstract Data Types (ADT) and extract candidate object models from the procedural code. Heuristic rules, metrics, and data flow analysis can be used to select an object model that is the most appropriate in a given context [1, 2]. Other methods to identify candidate classes from the procedural code, include concept analysis [5, 6], cluster analysis [3, 4], slicing [8], data flow and control flow analysis [9], and informal information analysis [7]. However, existing reengineering methods for migrating legacy systems to new object oriented platforms do not provide a comprehensive framework for ensuring that the migrant Object Oriented system will posses certain quality characteristics. To incorporate quality requirements into the migration process, we propose a reengineering approach that quantifies and assesses the impact each transformation step has on the target system. The specific quality requirements we focus in this paper are to increase modularity and cohesiveness for the new target system.

In this context, there are three major issues to be addressed namely, modeling the quality requirements for the target system, modeling the transformations in a formal way so that their preconditions and their impact on specific target quality requirements can be easily measured and evaluated and finally, a reengineering process that applies transformations in order to achieve the desired target qualities with a high likelihood score. For this work we adopt the non-functional requirement framework presented in [10] whereby software qualities and design decisions are modeled as a soft-goal dependency graphs. Similarly, software transformations are modeled as class templates in UML while their pre- and post- conditions that yield instantiations of transformation rules are modeled in OCL. Finally, the migration process is conceptually modeled as a Markov model of sequences of transformations whereby each transformation alters the state of the system by a given likelihood. The initial state corresponds to the original procedural system and the final state corresponds to the target migrant object oriented system. Given that each transformation alters at least one source code feature, a likelihood score as a measure of belief that the specific transformation impacts a specific quality can be computed. The likelihood score that a transformation contributes towards achieving a desired quality property is computed as a function of the source code features altered by the specific transformation and the relation of the altered source code features with the specific quality of interest.

The paper is organized as follows. Section 2 introduces in more detail the goal driven migration framework. Section 3 presents the formalization of transformation rules. Section 4 lists a catalogue of transformation rules that aim to extract a quality object model from the procedural source code. Section 5 discusses a migration process for the selection and combination of transformations. Section 6 presents an experiment for the transformation of selected gnu AVL libraries and evaluates the proposed method. Finally, section 7 concludes the paper and provides insights for future work.

2. Goal Driven Migration Process

In this reengineering framework we focus on the identification and extraction of quality object models from legacy procedural code. The specific qualities we consider for the new migrant system to posses are modularity and cohesiveness. The proposed goal driven migration process consists of several steps as illustrated in the Figure 1.

In our work, the subject software system is modeled in terms of entities and relations. The software entities represent source code structures of interest, such as aggregate types, parameters, global variables, and functions. The relations, model interactions between software entities, such as function calls, global variable usage, or aggregate data type references.

Essentially, the migration process either creates new software entities, for example a new class, or alters the associations between software entities that refer to the target object model. A set of possible transformation rules is identified to perform such re-architecture. Transformation rules are modeled as OCL expressions and can be sequentially composed to form a full migration process path. Assuming that each transformation alters the state of the system towards achieving the quality requirements of new target system, each transformation is selected according to its likelihood altering the state towards the desired requirements. The effect each transformation has on specific software qualities is modeled as a collection of soft goal dependency graphs. Soft-goal dependency graphs are directed labeled graphs that aim to associate design decisions with specific system non-functional requirements such as maintainability, performance, and reliability. The nodes of the graph represent interim qualities that need be achieved for the parent qualities to be achieved. Leaves of the graph represent specific design decisions and source code features that need be altered in order to achieve specific non-functional requirements. Applying a search algorithm such as A* or simulated annealing the order of transformations that achieve the highest likelihood the desired target system qualities for the migrant system can be computed.

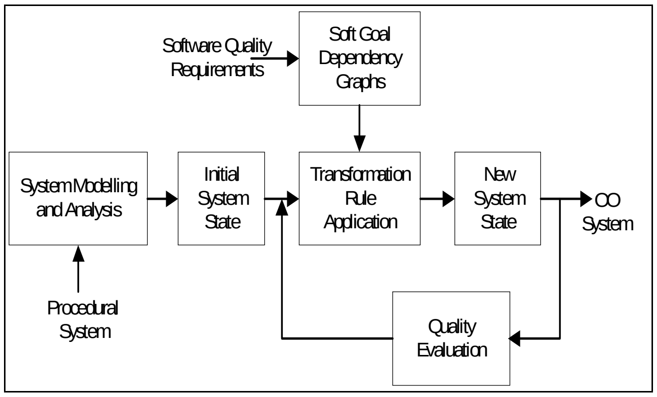

Software Quality Requirements
Soft Goal Dependency Graphs
System Modelling and Analysis
Initial System State
Transformation Rule Application
New System State
OO System
Procedural System
Quality Evaluation

Figure 1. Soft goal driven migration process

3. Specification of Transformations

In the proposed framework, transformation rules provide the means to implement generic migration steps. A transformation can be modeled as class template that has specific pre/post conditions and is instantiated for the source code context that it is applied upon. For example, a transformation template can be considered to be a rule that transforms global aggregate data types into candidate classes. We model transformation patterns as UML classes and pre/post conditions that govern the applicability of a given rule in a given context as OCL expressions. The following sections present these models.

3.1. UML and OCL Representation

The UML (Unified Modeling Language) provides a formalism for specifying, visualizing and documenting object-oriented systems in the form of class diagrams, object diagrams, use case diagrams, and state diagrams.

Furthermore, UML is enhanced by OCL (Object Constraint Language) that specifies well-formed constraints for the UML artifacts. Although OCL [15] is a formal language based on predicate logic and set theory, it provides a collection of textual expressions, such as types and operations that allow for articulating the invariants, pre/post-conditions of methods, and guards transition constraints in state transition diagrams.

In our research, it is of importance that the transformation rules can be interpreted without ambiguity. Due to its simplicity and formality, we believe that OCL is suitable candidate for the specification of software transformation rules. Furthermore, a rigorous proof for the correctness of the rules can be performed with the aid of the predicate logic and set theory.

3.2. OCL Expressions for Transformations

In general, a transformation rule template can be considered as a mapping function between a specific procedural source code feature and target object oriented code constructs. For example, an aggregate data type in the legacy system will be mapped to a candidate class in the new object oriented system. In this context, pre-conditions in OCL expressions denote the domain of the mapping function and the post-condition denote the range of the mapping function. An example OCL specification is illustrated below.

```
context ATypeName::OpName(parameter:Type1,
                                 …):AReturnType
pre:   parameter1> …
post:  result= expressions …
```

`context` is the keyword that declares that the specification of the operation, `OpName`, is in the context of the class (i.e. `ATypeName`). `pre` is the keyword for denoting pre-condition expressions, and so is the keyword `post` for the post-conditions. The `result` is the keyword that denotes the return value of the operation. OCL can model mathematical expressions using a collection of predefined operators such as, existential and universal qualification, iteration, and accumulation of values to a fix point by applying a specific expression to each element in a collection of elements.

3.3. UML Software Transformation Models

In this section, we present a collection of transformation rules that can assist in the extraction of an object model from a procedural system that is denoted as UML and OCL artifacts. In this context, there are two major categories of transformations. The first category aims to generate a class candidate from `Global Variables`, `Declarations`, and `Type_Specifiers`. This category is further subdivided to transformations that aim to generate the `Attribute` structure of the class candidate (i.e. its private data members), and to transformations that aim to generate the `Behaviour` of the candidate class (i.e. its methods). The second category of transformations aims to generate object oriented extensions to the simple model that can be extracted by simply analyzing global variables, parameters, and aggregate data types. The extensions deal with the introduction of Polymorphism, Inheritance and Overloading in the generated object model. The later is denoted by the `HierarchyTrans` and `PolymorphismTrans` classes in Figure 2, where the UML class diagram illustrates the static structure of a set of transformation rules.

The association between the `Attribute` and `Declaration` classes indicates that the attributes in class candidates are generated by the field declarations in the original code. In this context, the operations in the `AttributeGen` association class are referred to as

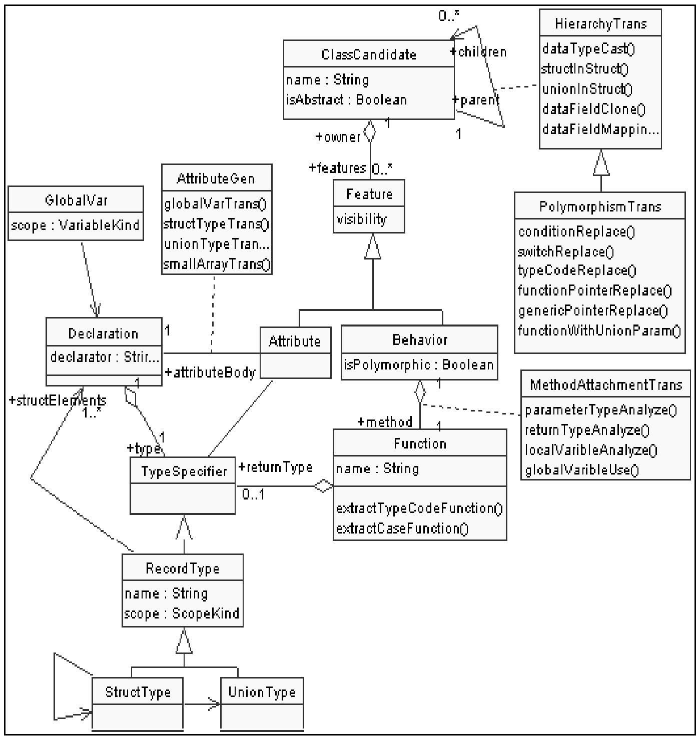


Figure 2. Migration process UML model

transformation templates and are responsible to denote the operations that migrate procedural field declarations of procedural aggregate `struct`, `record`, and `union` types to data members of candidate classes in the new object model. Furthermore, the detailed criterion for each transformation template is specified in OCL.

Finally, the association between the `Behavior` class and the `Function` class denotes the generation of methods from functions and procedures of the original legacy source code. For example, the association class, `MethodAttachmentTrans` provides four rules for function assignments in the corresponding operations.

4. Catalogue of Transformation Templates

In the migration process, it is important that the transformation rules not only yield object models, but also result in a new object oriented software system that possesses specific software qualities, such as high modularity, high cohesion inside the class and low coupling between classes. With the consideration of these goals, we identified and formally specified a series of transformation templates into two categories namely, class creation transformations and transformations for object model extensions. In the following sub-sections, we discuss the transformation rules and their OCL formal specification in more detail.

```
context AttributeGen::structTypeTrans(
            struct: StructType): Set(ClassCandidate)
pre: struct.nestingLevel = 1
post: result->including(adt:ClassCandidate |
            adt.name = struct.name and
            struct.structElements->iterate( elem: Declaration |
                        adt.features->including( attr:Attribute |
                                    attr.visibility = EnumType::private and
                                    attr.body = elem
                        )
            )
      )
```

Figure 3. Converting `Struct`Type into class candidate

4.1. Class Creation Transformations

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 that can be extracted from procedural source code, we aim at achieving high encapsulation, high cohesion within a class, and low coupling between classes. The process is divided into three steps: class identification, private data member identification and method attachment. The following sections provide indicative transformations that can be applied in each step.

4.1.1. Class Identification

The first step towards the migration of a procedural system to an object-oriented system is the selection of possible object classes. This task can be automated to a large extend using a number of different software analysis techniques. However, no matter how sophisticated the analysis techniques are, user assistance and guidance is crucial on obtaining a viable and efficient object model. Significant domain information can be utilized by the user to guide the discovery process and to obtain a better and more suitable object model. The object identification techniques focus on two areas: a) the analysis of global variables and their data types, b) the analysis of complex data types in formal parameter lists. Analysis of global variables and their corresponding data types is focusing 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 is focusing 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.

```
context AttributeGen::globalVarTrans(
            var : GlobalVar): Set(ClassCandidate)
pre: var.scope = EnumType::file or EnumType::globe
post: result->including(adt:ClassCandidate|
            adt.name=var.name and
            adt.features->including( attr: Attribute |
                        attr.visibility = EnumType::private and
                        attr.body = var
            )
      )
```

Figure 4. Converting global variable into class candidate

4.1.2. Private Data Member Identification

Data type analysis

Aggregate data types refer to a collection of data members inside a user-defined source code structure, such as `struct` and `union` in C. The OCL expression for the transformation of `struct` type is illustrated in Figure 3. The pre-condition requires that the `struct` type is not defined inside any other `struct` type. Since such `struct` type is globally available to be referenced by functions and can be used by other declarations throughout the program, it is will be suitable to be a class candidate in the new system. The post-condition characterizes the result of the transformation that all of the data members of the `struct` type become the private class attributes. Similarly, the `union` type can be converted into class candidate with the pre-condition that it is not embedded inside any other `struct` type definitions.

Variable analysis

Although C++ allows for global constant definitions to be accessible within file and global scope, keeping these scopes of variables unchanged in the new system would violate the principles of encapsulation and information hiding in the target object oriented system. The OCL expression in Figure 4, `globalVarTrans`, aims at eliminating such extensive scopes, by encapsulating such declarations as a private data member in an individual class.

4.1.3. Method Attachment

Parameter type analysis

A formal parameter in a procedure or a function indicates that the function references a data item of a particular type. In the process of object model extraction, we consider procedures and functions as method candidates. To maximize the cohesion inside the class and minimize the coupling between classes, the procedures and the function with `struct` parameter types are attached to the

```
context MethodAttachmentTrans::parameterTypeAnalyze(
              func: Function,
              adts: Set(ClassCandidate)) : Set(ClassCandidate)
pre:          func.parameters->size >=1
pre:          func.parameters->exists(param: Paramter|
                   param.type.oclType(StructType)=true and
                   adts->exists(adt: ClassCandidate|
                            adt.name= param.type.name))
post:         adts->iterate(adt:ClassCandidate;
                  result:Set(ClassCandidate)=Set{} |
                  func.parameters->iterate(param:Parameter|
                       if    param.type.oclType(StructType)= true and
                            param.name = adt.name
                       then
                            adt.features->including(op:Behavior|
                                  op.visibility= EnumType::public and
                                  op.method = func) and
                                  result->including(adt)
                       else result->including(adt)
                       endif
                  )
              )
```

Figure 5. Method attachment based on parameter type

```
context MethodAttachmentTrans::returnTypeAnalyze(
              func: Function,
              adts: Set(ClassCandidate)
              ): Set(ClassCandidate)
pre: func.returnType->notEmpty=true
pre: func.returnType.oclType(StructType)=true
pre: func.parameters->exists(param: Parameter |
              param.type.oclType(StructType)=true)
              ->size=0
post: adts->iterate(adt:ClassCandidate;
              result:Set(ClassCandidate)=Set{} |
                  if         adt.name= func.returnType.name and
                             adt->features->exists(op: Behavior|
                                op.method.name= func.name)->size=0
                  then       adt.features->including(op:Behavior|
                                op.visibility= EnumType::public and
                                op.method = func) and
                             result->including(adt)
                  else result->including(adt)
                  endif
        )
```

Figure 6. Method attachment based on return type

class candidates that are generated from these `struct` types. The formal expression for this transformation rule is illustrated in Figure 5.The pre-condition specifies the qualified function that has at least one parameter whose type is a `struct` type. The assignment of a function to a class is described in the post-condition that the function becomes a public method in more than one class candidates as the function can have more than one aggregate type parameters.

Return type analysis

The return type of a function indicates that the function possibly uses and/or updates the data fields of the aggregate type of the return value. Especially, in the case that a function without a parameter of an aggregate type, the return type provides strong evidence to assign such a function to the class candidate originated from the return type. Similar to the parameter type analysis, the transformation rule is illustrated in Figure 6.

Variable usage analysis

In the case that a function has neither aggregate type parameters, nor a return value of a aggregate type, the

```
context MethodAttachmentTrans::globalVariabeUse(
              func: Function,
              globalVars: Set(GlobalVar),
              adts: Set(ClassCandiate)):
              Set(ClassCandidate)
pre: func.parameters->exists(param: Parameter|
              param.type.oclType(StructType)=true)
              ->size = 0
pre: func.returnType->isEmpty=true or
              func.returnType.oclType(StructType)=false
pre: func.body.globalVars->size >= 1
post: adts->iterate(adt:ClassCandidate;
              result:Set(ClassCandidate)=Set{}|
              func.body.globalVars->iterate(var:GlobalVar |
                          if  var.declarator=adt.name and
                              adt->features->exists(op: Behavior|
                                 op.method.name = func.name)->size=0
                          then  adt.features->including(op:Behavior|
                                   op.visibility= EnumType::private and
                                   op.method = func) and
                                 result->including(adt)
                          else   result->including(adt)
                          endif
                          )
              )
```

Figure 7. Method attachment based on global variable usage in a function

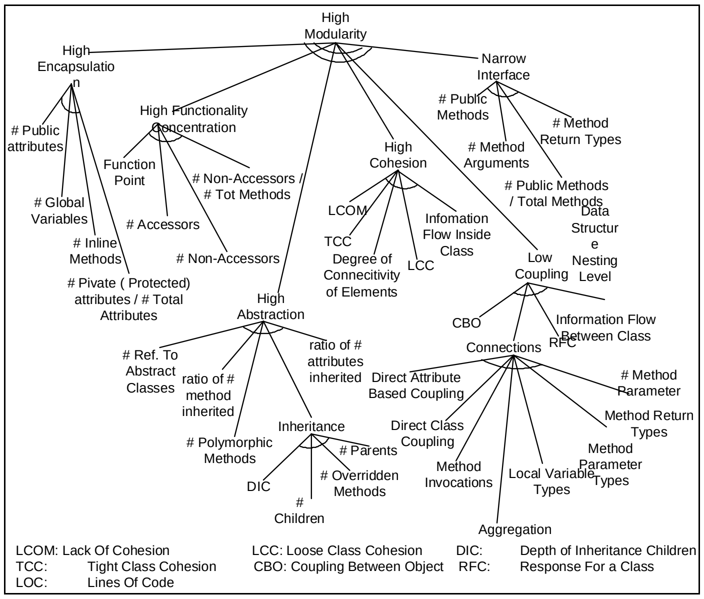

High Modularity
High Encapsulation
Narrow Interface
High Functionality Concentration
# Public Methods
# Method Return Types
# Public attributes
High Cohesion
# Method Arguments
Function Point
# Non-Accessors / # Tot Methods
# Public Methods / Total Methods
# Global Variables
# Accessors
LCOM
Infomation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

Figure 8. Soft-goal interdependency graph of high modularity

frequency of usage of aggregate types in the function body is considered as an evidence to transform the function to method in the class candidate that is generated by the aggregate type used. The pre-conditions and post-conditions of this transformation rule are illustrated in Figure 7.

5. Selection and Combination of Transformations

In [10], we proposed soft-goal dependency graphs that systematically model source code features that are related to reliability and maintainability. An example soft-goal interdependency graph is illustrated in Figure 8. The leaves of graph list the source code features that have impacts on high modularity. By such a graph, the changes in the source code features can be traced back to reflect the changes in the high-level software quality goals.

Moreover, in section 4, a sample of possible transformation templates is specified. We consider that the migration process can be modeled as a sequence of transformations that alters features identified in the soft-goal graph. Consequently, we consider that these transformations have an impact on the modeled quality (i.e. maintainability) when they are applied. The objective thus is to identify the combination of transformations that have the highest likelihood of achieving the specified quality requirements for the target migrant system. For this work, we adopt an approach that is based on a Markov model to denote the likelihood that a transformation when applied in one system state will yield a new system state with better quality characteristics.

5.1. Quality Driven Migration Process Model

Conceptually, the migration process can be modeled as a sequence of transformations in a labeled system state transition system [18]. A formal definition for a migration system is as follows.

Definition 1: A migration process is a tuple:

$$(S, I, F, T, \xrightarrow{t})$$

where:

- S is a non-empty states, s_0, s_1, …, s_i, s_{i+1}, …, s_n.
- *I* represents the original software system state.
- *F* represents a set of final states that corresponds to the resulting migrant system.
- T is a set of transformations, t_{01}, t_{02}, …, t_{ij}, $t_{i,j+1}$, …, t_{kn}, each of which alters a state and yields a consecutive state and aims to transform a software system in a stepwise faction from its initial state to a final state that correspond to the original system and new system. t_{ij} represents the transformation moving from s_i to s_i.
- $\xrightarrow{t} \subset S \times T \times S$ is a set of rules, which define the semantic meaning for transformations.

Definition 2: A feature vector v represents the quality with respect to a non-functional requirement in a state and is denoted by a set of attributes $<a_1, a_2, \dots, a_k, \dots, a_m>$, where a_k quantifies in a numeric format a source code feature, which is a terminal in soft goal interdependency graphs.

Definition 3: Two states are distinct if their feature vectors are different.

As presented above, system sate changes are achieved by the application of transformations from the set T and conform to the following rules:

Rule 1: Every transformation t_{ij} causes at least one change in a selected code feature that quantifies state s_i and results in state s_j.

Rule 2: The change is quantified by the identified source code features modeled as leaves of the soft-goal graphs.

As stated in Rule 1, a transformation makes changes to a state. A transformation may cause the value a_k to increase, decrease, or keep it the same. As a consequence, the change is quantified by a delta on the corresponding feature values. The more positive the impact is, the higher the likelihood that the transformation can contribute towards the desired quantity objectives. Therefore, a transformation is combined with a quality factor that is used to evaluate the quality contribution each transformation has.

The following formulae (1), (2) are proposed to compute the likelihood of $\lambda_{(G)ij}$,, called as quality factor, that the transformation t_{ij} improves the quality characteristics of the system with respect to the quality goal *G*. When the number of the changed features that contribute positively towards the desired quality is higher than the number of features changed that contribute negatively towards the desired quality, the following formula is used:

$$\lambda_{(G)ij} = \frac{\sum \text{Positve Impact} - \sum \text{Negative Impact}}{\sum \text{Attribute}} \qquad (1)$$

In the cases that the negative impacts are larger than positive changes, we take the logarithm of the result and the following formula (2) is applied:

$$\lambda_{(G)ij} = e^{\frac{\sum \text{Positve Impact} - \sum \text{Negative Impact}}{\sum \text{Attribute}}} \qquad (2)$$

It is also important to note that in many cases a goal is achieved if its sub-goals are also achieved. To compute the likelihood score of a goal as a composition of likelihood scores of its sub-goals, we propose the following formula (3). In addition, some sub-goals are more important than others and in this case goal weights are determined by the users, and are added as a coefficient c_k.

$$\lambda_{(G)ij} = e^{\sum_{k=1}^{m} c_k \lambda_{(k)ij}} \quad (3)$$

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).*

The above formula (3) can be applied recursively at different levels of the soft-goal dependency graphs. In addition, using the above formula, we can calculate the overall likelihood to achieve more than one quality objective. It is worth noting that the likelihood $\lambda_{(G)ij}$, only depends upon the immediately preceding states s_i, and not upon other previous states.

5.2. Transformation Composition

For the migration of procedural code to object oriented platform, we have previously identified a catalog of transformation rules to apply at the procedural in order to extract an object-oriented model [2]. A subset of the transformation rules is described in terms of pre/post conditions, as shown in Table 1.

To model all possible transformation paths the stochastic process algebra formalism is adopted. As illustrated in Figure 9, an example specification is as follows:

$S_0 = (R_1, \lambda_{01}).S_1 + (R_2, \lambda_{02}).S_2$
$S_1 = (R_3, \lambda_{13}).S_3$, $S_3 = (R_4, \lambda_{35}).S_5$, $S_5 = (R_5, \lambda_{57}).S_7$
$S_2 = (R_5, \lambda_{24}).S_4$, $S_4 = (R_3, \lambda_{46}).S_6$, $S_6 = (R_4, \lambda_{67}).S_7$
S_7 = exit

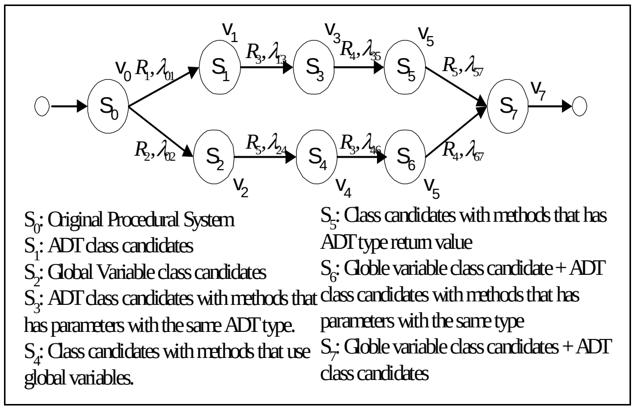
where, the symbol, "=", is used to assign names to processes, and the symbol, "+", represents the process behaves either S_1 or S_2, and the choice is determined by the value of λ_{01} and λ_{02}. Similarly, the symbol "." represents the prefix of processes. For example, $(R_3, \lambda_{13}).S_3$ means the process engages in a transformation under the rule R_3 with the quality factor λ_{13} and subsequently behaves as S_3. Such a specification discloses the general behaviors of the migration process by the use of rules to label the transitions. A concrete system evolution is generated by applying the transformations with the conformant to the rules.

5.3. Optimal Transformation Path

The objective of the migration process is to compute an optimal transformation path that can yield a target system that meets specific quality requirements. Similar to labeled transition systems, Markov chains are directed graphs where the transitions are labeled by probability scores. A simple example is shown in Figure 10. The probability predicts the likelihood that a transition can happen. It is straightforward that a stochastic process can be mapped to a Markov chain by deriving the probability for each transition [18]. In our work, the probability for

Table 1: A Subset of Transformation Rules for Object Oriented Migration

Rule	Pre-condition	Post-condition
R_1	Aggregate Data Types (ADTs) are class candidates	Transforms each data field in ADTs into attributes in the corresponding classes
R_2	Each global variables are encapsulated as class candidates	Transforms each global variable into the attribute in the corresponding classes
R_3	Function has the parameters with aggregated data type (ADT)s	Attach this function to the class that is created from the ADT of the parameter type
R_4	Function, w/o parameters of ADT, has a return value with an ADT	Attach such a function to the class that is created from the ADT of the return value
R_5	Function, w/o parameters and return value, makes use of globe variables	Attach such function to the class that is created from the global variable.



S_0: Original Procedural System
S_1: ADT class candidates
S_2: Global Variable class candidates
S_3: ADT class candidates with methods that has parameters with the same ADT type.
S_4: Class candidates with methods that use global variables.
S_5: Class candidates with methods that has ADT type return value
S_6: Globle variable class candidate + ADT class candidates with methods that has parameters with the same type
S_7: Globle variable class candidates + ADT class candidates

Figure 9. State Evolution For Identifying Class Candidates from Procedural Code

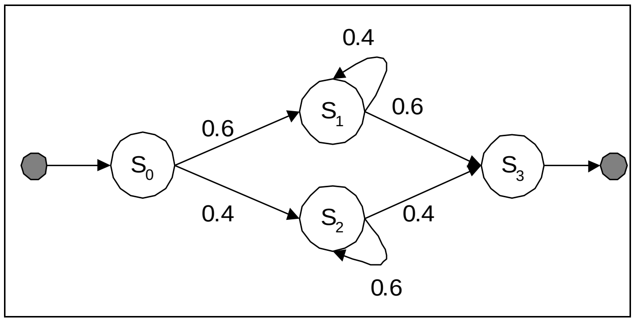


Figure 10. A Markov Model Example

each transition is calculated by formula (4) and (5). In the case that multiple transformation alternatives be triggered from the same state, the probability can be calculated by the formula (4). For the sequential transformations, the probability is defined by the formula (5):

$$q_{ij} = \frac{\lambda_{ij}}{\sum_{k=n}^{m} \lambda_{ik}}, \text{ when multiple states evolve from } S_i \qquad (4)$$

$$q_{ij} = \frac{\lambda_{ij}}{1+\lambda_{ij}}, \text{ when one state is followed from } S_i \qquad (5)$$

where λ_{ij} is the quality factor for a transformation between S_i and S_j, n is the smallest index of the state that follows S_i, and m is the largest index of the state that moves from S_i.

In this context, the larger the quality factor is, the higher likelihood such transformation results in a better quality state. Based on the Markov chain approach, the likelihood of different transformation paths can be calculated. To get the path with the highest likelihood that reaches desired goals, the Viterbi algorithm [19] is used.

6. Experiments

To investigate the feasibility of such a quality driven re-engineering framework, we apply it for the migration of the *gnu* AVL tree libraries from its original procedural implementation to an object oriented one. For the experimentation purposes of this paper, we use the transformation rules listed in Table 1 to extract an object model from the gnu AVL system. The whole migration process is an instantiation of the general model (shown in Figure 8) that gives the constraints and imposes orders to apply the transformation rules based on the pre and post conditions of the rules.

6.1. Quality Goals and Metric Collection

For this case study, the target requirements for the new system are to achieve high encapsulation, as well as high cohesion and low coupling. These quality attributes can be considered as sub-goals, and consequently achieve higher-level soft-goals for the new system such as high encapsulation, high cohesion, low coupling, reusability and maintainability. For each of the soft-goals, a set of metrics was considered, as illustrated in Figure 11.

```
Encapsulation <NPA, NGV, PAR>, where
     NPA: Number of Public Attribute    NGV: Number of Global
          Variable
     PAR: Private Attributes Ratio
Cohesion <IFIC>, where
     IFIC: Information Flow Inside Class
Coupling <CBO, IFBC, DCC, NMI, NLVT, NMPT, NMRT>, where
     CBO: Coupling between Objects IFBC: Information Flow
          Between Classes
     DCC: Direct Class Coupling (count of the different
          number of classes that a class is directly related
          by attribute declarations and parameters in
          methods.)
     NMI:  Number of Method Invocations in other classes
     NLVT: Number of Local Variable Types from other classes
     NMPT: Number of Method Parameter Types from other
           classes
     NMRT: Number of Method Return Types from other classes.
```

Figure 11. Software Goals and Metric Sets

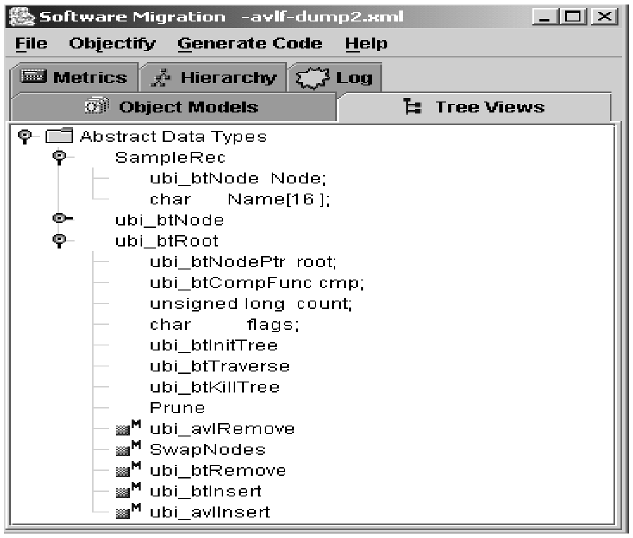
Table 2: Encapsulation Measurement on Converting `Struct` Type into Class Candidate

	Sample Rec	**ubi_ btNode**	**ubi_ btRoot**
NPA	+2	+3	+4
NGV	-	-	-
PAR	+1	+1	+1
$\lambda_{(encapsulation)ij}$ (Formula 1)	0.6667	0.6667	0.6667
$\lambda_{(encapsulation)ij}$ (Formula 2)	1.9478	1.9478	1.9478

6.2. Transformations and State Evolutions

As specified in the stochastic process algebras, the migration process firstly aims to achieve high abstraction where the global variables and global aggregated data types are converted into class candidates. In AVL systems, there are three global aggregated data types, including `SampleRec`, `ubi_btNode` and `ubi_btRoot`. The quality factors for each of the transformations that convert a data type into candidate classes is illustrated in Table 2.The values in the table cells from the row 2 to row 4 illustrate the changes in source code features after applying the transformations to identify classes. Formula (1) and (2) are applied to compute the quality factors towards encapsulation. All the transformations contribute towards positive feature changes, and therefore contribute positively towards the desired quality with the same likelihood; hence the transformations can be selected in any order.

The initial break down of the system is achieved and illustrated in Figure 12. The methods with square labels denote the potential methods that can be attached to more than one class. All these methods can be assigned either


Software Migration -avlf-dump2.xml
File Objectify Generate Code Help
Metrics Hierarchy Log
Object Models Tree Views

```
Abstract Data Types
    SampleRec
        ubi_btNode  Node;
        char     Name[16];
    ubi_btNode
    ubi_btRoot
        ubi_btNodePtr root;
        ubi_btCompFunc cmp;
        unsigned long  count;
        char        flags;
        ubi_btInitTree
        ubi_btTraverse
        ubi_btKillTree
        Prune
        ubi_avlRemove
        SwapNodes
        ubi_btRemove
        ubi_btInsert
        ubi_avlInsert
```

Figure 12. System State with Initial Classes

Table 3: Coupling measurement for resolving the attachment of method `avl_btInsert` to a class

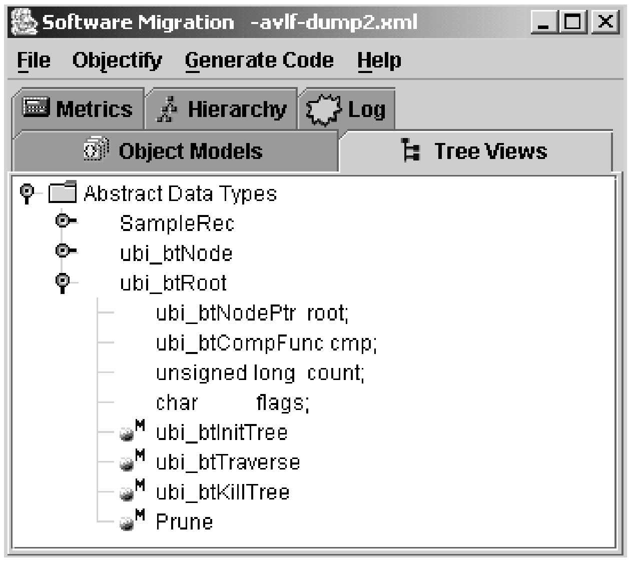
Assigned Class	**Ubi_btNode**	**ubi_btRoot**
CBO	-9	-11
IFBC	0	-11
DCC	-1	-2
NMI	0	-4
NLVT	0	-1
NMPT	-1	-1
NMRT	0	-1
$\lambda_{(coupling)ij}$ (Formula 1)	-0.4286	-1
$\lambda_{(coupling)ij}$ (Formula 2)	0.6514	0.3679

Table 4: Cohesion measurement for resolving the attachment of method `avl_btInsert` to a class

Assigned Class	**ubi_btNode**	**ubi_btRoot**
IFIC	+11	0
$\lambda_{(cohesion)ij}$ (Formula 1)	1	0
$\lambda_{(cohesion)ij}$ (Formula 2)	2.7183	1

Table 5: Accumulative result for resolving the attachment of method `avl_btInsert` to a class

Assigned Class	**ubi_btNode**	**ubi_btRoot**
λ_{ij} (Formula 3)	λ_{69} 29.0697	$\lambda_{(6,10)}$ 3.9271


Software Migration -avlf-dump2.xml
File Objectify Generate Code Help
Metrics Hierarchy Log
Object Models Tree Views

```
Abstract Data Types
    SampleRec
    ubi_btNode
    ubi_btRoot
        ubi_btNodePtr root;
        ubi_btCompFunc cmp;
        unsigned long  count;
        char       flags;
        ubi_btInitTree
        ubi_btTraverse
        ubi_btKillTree
        Prune
```

Figure 13: System State without Method Conflicts

to `ubi_btRoot` or `ubi_btNode`. Table 3 illustrates the changes of the features related to coupling, if the method `avl_btInsert` is assigned to either class. The values in the table cells from the row 2 to row 8 illustrate the deltas of the source code features between two consecutive states. According to formulas 1 and 2, the cases of $\lambda_{(coupling)ij}$ are calculated, respectively. Similarly, table 4 illustrates the impact on cohesion. Finally by utilizing formula 3, the cumulative result of the impact on both goals is calculated, as shown in Table 5. Thus, the `avl_btInsert` is assigned to `ubi_btNode`, because it has higher likelihood according to the impacted features, to achieve the desired software goals. The rest of the conflicting methods can be resolved in the same way. Figure 13 illustrates the state where all classes have been identified and no methods are in conflict.

In addition, the selection of consecutive transformations is not only determined by the corresponding quality factors, but also the internal dependencies of the methods. For example, the method, `ubi_avlRemove`, depends on the method, `ubi_btRemove`, which depends on the method, `SwapNode`. Therefore, the resolution of the method, `SwapNode`, is crucial for the other two methods. Similarly, the method, `ubi_avlInsert` depends on the method, `ubi_btInsert`.

Finally, formulae (4) and (5) can be used to obtain an optimal transformation path in a Markov transformation chain. The final state of the system is illustrated in Figure 14 as a UML diagram.

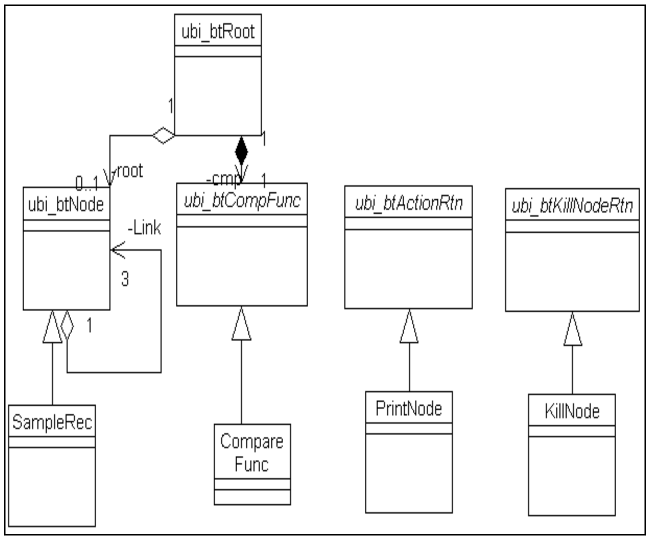


Figure 14. Final Retrieved Object Model

7. Conclusion

This paper presents a quality driven reengineering framework that constructs the migration process as a labeled state transition system, and evaluates the fulfillment of soft quality goals at each step of the process. The framework is characterized by four sub-models, including an object model, in which states are represented as entities and relations, a transformation model, in which transformation rules are formally specified in terms of pre/post conditions, a software quality model, in which the specific quality features can be traced to source code features, and a migration process model that selects and composes transformations based on their contribution to the desired qualities. Moreover, the migration process is formally specified by stochastic process algebra. In this context, the software quality model is incorporated into the migration process by the associating each transformation with a quality factor. By the use of stochastic process algebra, a Markov chain is automatically generated and facilitates to find an optimal transformation path to achieve a desired system.

Currently, the proposed framework is applied to migrate systems written in C to functionally similar systems that comply with an object oriented design and implemented in C++. On-going work is focusing on generating soft-goal graphs for portability and testability and applying the framework for the migration of larger than 4KOC systems.

References

[1] Lionel Briand, et. al, “Characterizing and Accessing a Large-Scale Software Maintenance Organization”, http://www.cs.umd.edu/projects/SoftEng/ESEG/

[2] Ying Zou, Kostas Kontogiannis, “A Framework for Migrating Procedural Code to Object Oriented Platform”, in the proceedings of 8th Asia-Pacific Software Engineering Conference, Macau SAR, China, December 4-7, 2001.

[3] S. Mancoridis, B.S. Mitchell, Y. Chen, and E. R. Gansner, Bunch: a clustering tool for the recovery and maintenance of software system structures, In Proc. Of International Conference on Software Engineering, 1999.

[4] H. Muller, M. Orgun, S. Tilley, and J.Uhl, A reverse Engineering Approach To Subsystem Structure Identification, In Journal of Software Maintenance: Research and Practive, 5(4): 181-204, 1993.

[5] 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.

[6] 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.

[7] Letha H. Etzkorn, Carl G. Davis, “Automatically Identifying Reusable OO Legacy Code”, Computer, IEEE, October, 1997.

[8] Filippo Lanubile, and Giuseppe Visaggio, “Extracting Reusable Functions by Flow Graph-Based Program Slicing”, IEEE Transactions on Software Engineering, Vol. 23, No. 4, April, 1997.

[9] De Lucia, G.A. Di Lucca, A.R. Fasolino, P. Guerra, S. Petruzzelli, “Migrating Legacy Systems toward Object Oriented Platforms”, 1997, IEEE.

[10] Ying Zou, Kostas Kontogiannis, “Migration to Object Oriented Platforms: A State Transformation Approach”, International Conference on Software Maintenance 2002.

[11] Ladan Tahvildari, Kostas Kontogiannis, John Mylopoulos, “Requirements-Driven Software Reengineering", *8th IEEE Working Conference on Reverse Engineering (WCRE 2001)*, Stuttgart, Germany, pp. 71-80, October 2001.

[12] Ladan Tahvildari, Kostas Kontogiannis, “On the role of design patterns in quality-driven re-engineering”, In Proceedings of the 6th IEEE European Conference on Software Maintenance and Re-engineering (CSMR), Hungary, Budapest, march 2002.

[13] K. Kontogiannis, P. Patil, “Evidence Driven Object Identification in Procedural Systems’’. STEP’99, September 1999, pp. 12-21.

[14] International Standard for Software Product Quality Software (ISO/IEC 9126: 1991).

[15] “OMG Unified Modeling Language Specification”, ftp://ftp.omg.org/pub/docs/formal/01-09-67.pdf.

[16] Martin Fowler, “Refactoring: Improving the Design of Existing Code”, Addison-Wesley, 2000.

[17] Stuart Russell, et. al, “Artifical Intelligence, A Modern Approach”, Englewood Cliffs, N.J. : Prentice Hall, 1995.

[18] Ed Brinksma and Holger Hermanns, “Process Algebra and Markov Chains”, FMPA 2000, LNCS 2090, pp.183-231, 2001, Springer-Verlag Berlin Heidelberg 2001.

[19] Paul van Alphen & Dick R. van Bergem, “Markov Models and Their Application in Speech Recognition”.