Quality Driven Transformation Framework 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 that produces a platform for network-centric application integration. 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 (Object Constraint Language). Finally, the migration process is conceptually modeled as a sequence 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 techniques for quality driven reengineering.

Parts of this paper appear in the IEEE -ICSM 2002 and IEEE-APSEC 2002 proceedings. Reprinted with permission from the IEEE Computer Society. © IEEE Computer Society.

Section 3 presents the formalization of transformation rules that aim to extract a quality object model from the procedural source code. Section 4 discusses a migration process for the selection and combination of transformations. Section 5 presents an experiment for the transformation of selected gnu AVL libraries. Finally, section 6 concludes the paper and provides insights for future work.

2. Quality Driven Reengineering

In this section, we further discuss the concept of quality driven reengineering and techniques that can be used to build the quality driven migration process as originally presented in [11, 12]. These techniques include the classification of software quality characteristics, soft-goal dependency graphs, and software metrics. The objective of quality-driven reengineering is to provide a framework whereby the migration process is tailored towards achieving specific requirements for the migrant system.

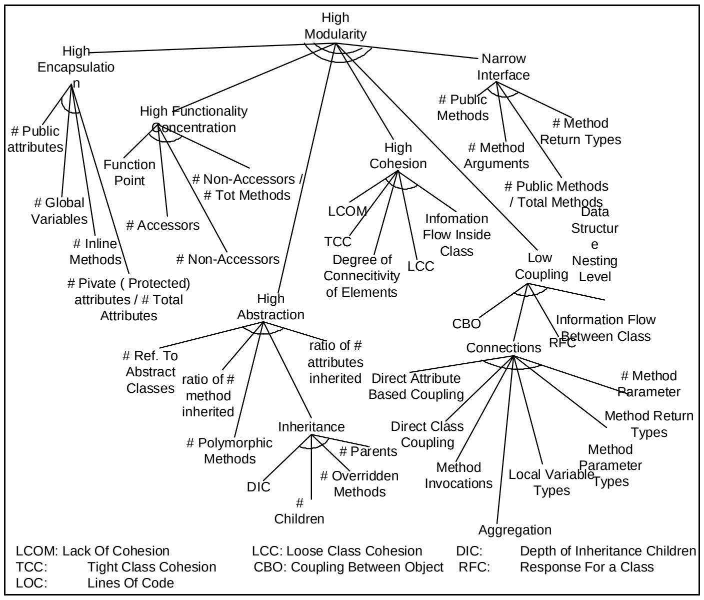
2.1. Software Quality Characteristics

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.

The International Standard Organization for Software Product Quality Software (ISO/IEC 9126: 1991(E)) has identified six main external attributes [13] namely: functionality, reliability, usability, maintainability, portability and efficiency. More recently, another external quality attribute that has received attention especially because of the widespread use of the object oriented technology, is reusability. Each of the attributes is further subdivided in different sub-categories. For example, maintainability is further subdivided into analyzability, changeability, stability and testability [13].

2.2. Soft-Goal Dependency Graphs

A soft-goal dependency graph is a graph composed of nodes and edges. Nodes represent goals to be satisfied in order to achieve a desired quality property. Edges represent dependencies as to how these goals can be satisfied. The term soft-goal refers to the property of the



High Modularity
High Encapsulation
High Functionality Concentration
Narrow Interface
# Public attributes
# Global Variables
# Inline Methods
# Pivate (Protected) attributes / # Total Attributes
Function Point
# Accessors
# Non-Accessors / # Tot Methods
# Non-Accessors
High Cohesion
LCOM
TCC
Degree of Connecitivity of Elements
LCC
Infomation Flow Inside Class
# Public Methods
# Method Arguments
# Method Return Types
# Public Methods / Total Methods
Data Structure Nesting Level
Low Coupling
CBO
RFC
Information Flow Between Class
Connections
Direct Attribute Based Coupling
Direct Class Coupling
Method Invocations
Aggregation
Local Variable Types
Method Parameter Types
Method Return Types
# Method Parameter
High Abstraction
# Ref. To Abstract Classes
ratio of # method inherited
ratio of # attributes inherited
# Polymorphic Methods
Inheritance
DIC
# Children
# Overridden Methods
# Parents
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 1. Soft-goal interdependency graph of high modularity

graph that dependencies to its sub-goals may be also satisfied partially for the parent goal to succeed, and that nodes are used to capture informal concepts [14]. For example, Figure 1 illustrates soft-goal graphs related to high modularity, which can be achieved by the sub-goals, such as high encapsulation, high abstraction, high cohesion and low coupling. Soft-goals may depend on sub-goals according to AND/OR relations. An AND dependency means that all sub-goals need be satisfied 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.

2.3. Quality Measurement

For each source code quality modeled in a soft-goal graph, a set of metrics and the corresponding source code features used to compute these metrics are selected. These features appear as leaves in the soft-goal dependency graph. The magnitude of change due to a re-engineering transformation provides an indicator on the magnitude of change in the corresponding metric and therefore in the corresponding quality modeled by the graph. For example, in Figure 1, 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.

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 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. Selection and Combination of Transformations

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.

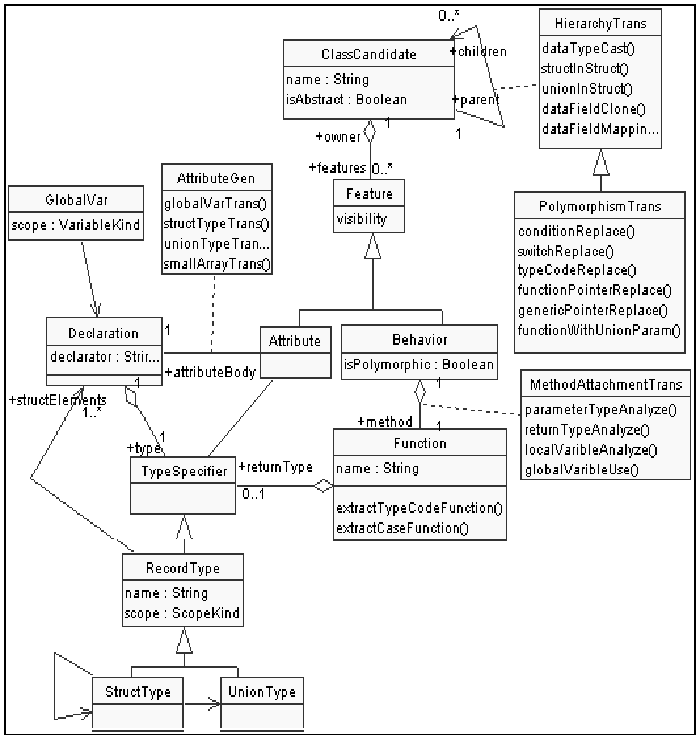


Figure 2. Migration process UML model

4.1. Quality Driven Migration Process Model

Conceptually, the migration process can be modeled as a sequence of transformations in a labeled system state transition system [16]. 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, …, a_k, …,

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:

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

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:

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

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.

4.2. Transformation Composition

For the migration of procedural code to object oriented platform, we have 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 3, 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

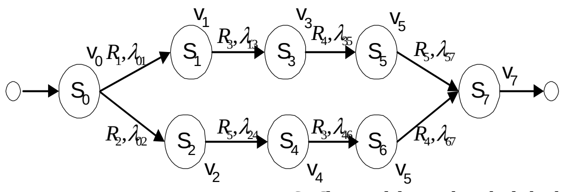
where, the symbol, "=", is used to assign names to processes, and the symbol, "+", represents the process behaves either S_1 or S_2, and the choice is determined by the value of λ_{01} and λ_{02}. Similarly, the symbol "." represents the prefix of processes. For example, $(R_3, \lambda_{13}).S_3$ means 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.

4.3. Optimal Transformation Path

The objective of the migration process is to compute an optimal transformation path that can yield a target system

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 3. State Evolution For Identifying Class Candidates from Procedural Code

that meets specific quality requirements. Similar to labeled transition systems, Markov chains are directed graphs where the transitions are labeled by probability scores. It is straightforward that a stochastic process can be mapped to a Markov chain by deriving the quality likelihood for each transition. 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 [10, 17] is used.

5. 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

```
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 4. Software Goals and Metric Sets

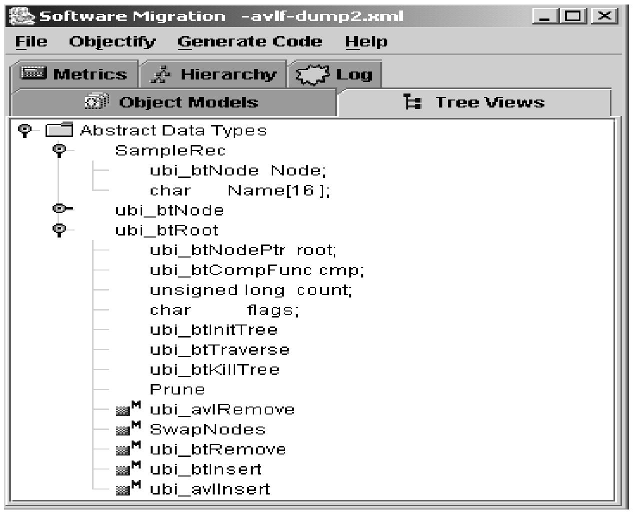
transformation rules listed in Table 1 to extract an object model from the gnu AVL system. The whole migration process is an instantiation of the general model (shown in Figure 3) that gives the constraints and imposes orders to apply the transformation rules based on the pre and post conditions of the rules.

5.1. 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 4.

5.2. 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 initial break down of the system is achieved and illustrated in Figure 5. 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 to `ubi_btRoot` or `ubi_btNode`. Table 2 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 3 illustrates the impact on cohesion. Finally by utilizing



```
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 5. System State with Initial Classes

Table 2: 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

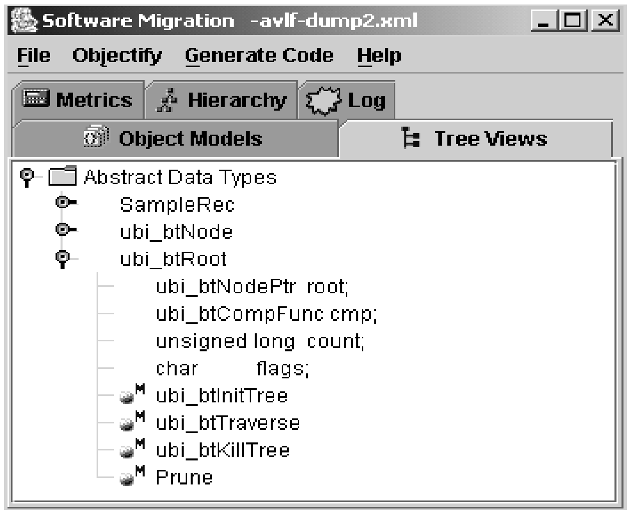
Table 3: Cohesion measurement for resolving the attachment of method `avl_btInsert` to a class

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 4: 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

formula 3, the cumulative result of the impact on both goals is calculated, as shown in Table 4. 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



```
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 6: System State without Method Conflicts

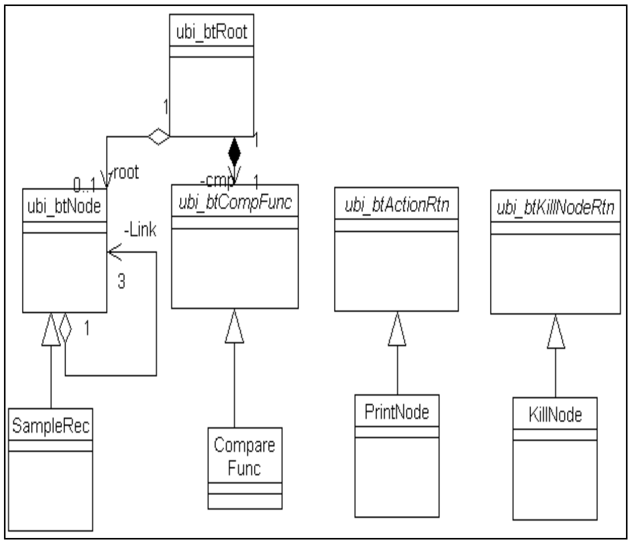


Figure 7. Final Retrieved Object Model

conflicting methods can be resolved in the same way. Figure 6 illustrates the state where all classes have been identified and no methods are in conflict. The final state of the system is illustrated in Figure 7 as a UML diagram.

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

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

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

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

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

[17] Paul van Alphen & Dick R. van Bergem, "Markov Models and Their Application in Speech Recognition".