Migration to Object Oriented Platforms: A State Transformation Approach

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

Abstract

Over the past years it has become evident that the benefits of object orientation warrant the design and development of reengineering methods that aim to migrate legacy procedural systems to modern object oriented platforms. However, most research efforts in this direction focus mostly on the extraction of an object model from the legacy procedural code without taking into account quality requirements for the target migrant system. This paper presents a reengineering workbench that allows for quality requirements for the target system to be modeled as soft-goals and software transformations to be applied selectively towards achieving specific quality requirements for the target system. In this context, the migration process is denoted by a sequence of transformations that alter the state of the system being reengineered. A Markov model approach and the Viterbi algorithm are used to identify the optimal sequence of transformations that can be applied at any given state of the migration process. For the evaluation of the proposed workbench, a migration experiment of the gnu AVL tree libraries is presented.

1. Introduction

Legacy systems refer to mission critical software systems that are still in operation, but their quality and expected operational life is constantly deteriorating due to prolonged maintenance and technology updates. To leverage the business value entailed in such systems, a possible solution is to migrate selected parts of such systems to modern platforms and designs. One such possible solution is related to object oriented re-engineering whereby a procedural legacy system or component is migrated towards a modern object oriented design. With properties, such as information hiding, inheritance and polymorphism inherent in object oriented designs, essential parts of such a reengineered system can be reused or integrated with other applications using network centric Web technologies, enterprise integration solutions, or even third generation networks.

In this context, the software reengineering community has already proposed a number of different methods to migrate procedural code into object oriented platforms. In a nutshell, the existing migration methods aim to identify Abstract Data Types (ADT) and extract candidate classes and methods from the procedural code. These methods include concept analysis [12, 13], cluster analysis [10, 11], slicing [16], data flow and control flow analysis [17], source code features [15], and informal information analysis [14]. However, existing reengineering methods towards object oriented platforms do not provide a comprehensive framework for ensuring that the migrant system will posses certain quality characteristics.

In this paper, we present a reengineering approach that monitors and evaluates the software quality of the system being reengineered at each stage of the migration process. Specifically, the migration process is denoted by a sequence of transformations that alter the state of the system. The initial state corresponds to the original system and the final state corresponds to the target migrant system. Each system-state is qualified by a feature vector that is associated with one or more target qualities (i.e. performance, maintainability). A soft-goal dependency graph is used to make this association of features and software qualities explicit. In this way, the reengineering workbench aims to provide a comprehensive framework whereby possible transformations can be selected, applied, and assessed towards achieving the desired target qualities. Therefore, the research problems to be addressed include three areas. First, the specification of the relationships of source code features with specific target qualities; second, the identification of a comprehensive set of possible transformations that can be applied to migrate a system to an object oriented platform; and finally, the design of an algorithmic process that can be applied to identify an optimal sequence of transformations that can yield the new target system given a specific legacy system. To achieve the first research objective, we consider the use of soft-goal graphs proposed within the context of Non-

Functional Requirements. Such graphs allow for modeling the impact of specific design decisions towards a specific target requirement [24, 26]. The leaves of such graphs correspond to concrete system attributes that impact all the other nodes to which they are connected. In such graphs, nodes represent design decisions, and edges denote positive or negative dependencies towards a specific requirement. The second research objective is addressed by devising a comprehensive, yet extensible list of transformations that can be used to migrate a procedural system to an object oriented one. We have identified a number of such transformations and we have presented them in related research papers [15, 25]. Finally, the third research objective is achieved by the utilization of Markov models and the Viterbi algorithm [9], whereby the optimal sequence of transformations towards achieving the desired qualities is identified. The selection of the transformations is based on quantitative methods and a probabilistic model that aims to quantify the magnitude by which deltas in specific source attributes after applying a transformation may contribute towards achieving a desired system property.

The rest of the paper is organized as follows. Section 2 introduces the software quality characteristics, soft-goal quality dependency graphs and software metrics. Section 3 discusses the dependencies between software quality and source code features. Section 4 describes the proposed migration process model. Section 5 provides a case study by applying the proposed approach to a medium size system. Finally, section 6 concludes the paper.

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 [24, 26]. 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 [18] 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 [18]. The following subsections discuss in more detail the soft-goal dependency graphs as means to model the dependencies between the software qualities and internal source code attributes.

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 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 [23]. For example, Figure 1 and 2 illustrate soft-goal graphs related to reusability and maintainability. In the example graph of Figure 1, reusability can be achieved by high modularity, low complexity, and good documentation. 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 [24].

2.3. Software Metrics

Software metrics provide measurements of certain characteristics of a software system that are valuable to the specification, design, and project management [19]. In a nutshell, software metrics can be classified into three major categories [19] namely, product metrics, process metrics, and project metrics. Product metrics are used to measure the internal and external characteristics of a system. These include information flow metrics, function point metrics, cyclomatic complexity metrics, and information science metrics. Process and project metrics are defined to measure the software development and maintenance life cycle. Examples of process and project metrics include productivity measurements, software

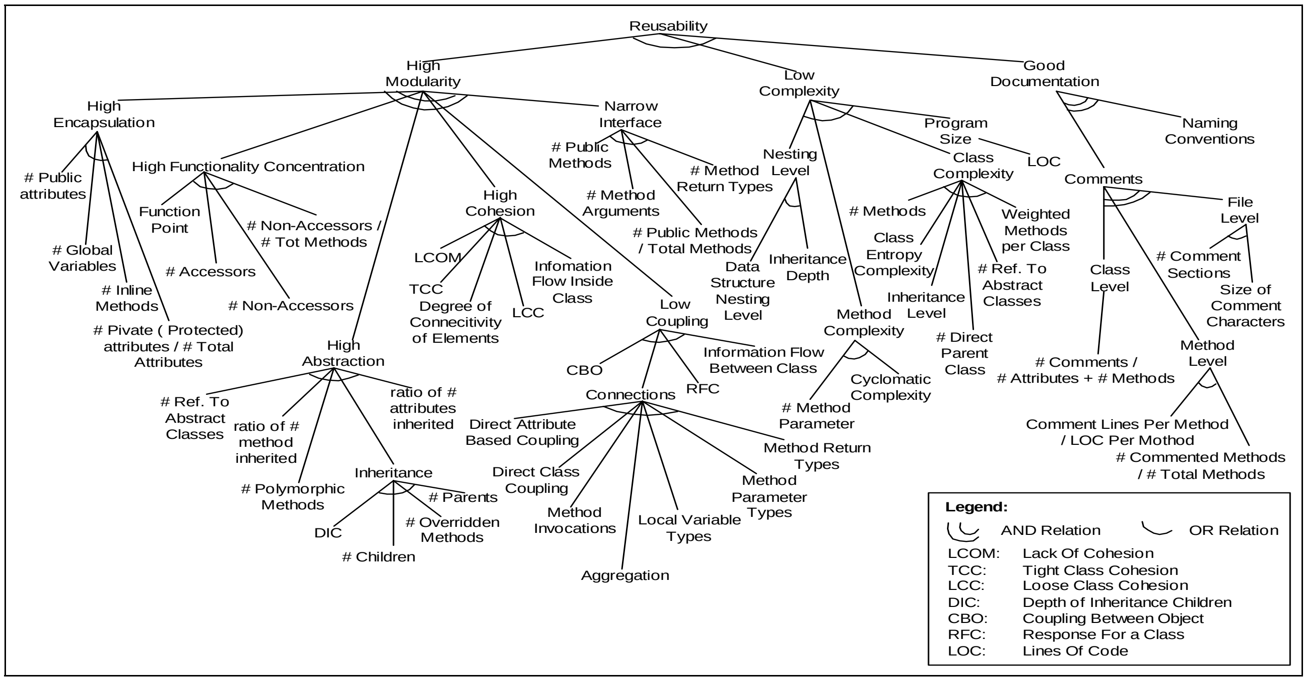


Figure 1: Reusability Soft-Goal Dependency Graph

size and effort prediction metrics, defect detection and removal effort metrics, as well as cost estimation metrics. In the context of our research, we are interested in examining a number of product metrics that are related to reusability and maintainability, and identifying a set of source code features that impact these metrics. The basic assumption is that the magnitude of change in these features directly relates to qualities that the metrics are measuring (i.e. reusability, maintainability). These features appear as leaves in the soft-goal dependency graphs and are discussed in more details in the following sections.

3. Software Quality Models

The major objectives of our study are to identify software features related to specific quality requirements, to model their inter-dependencies, to assess their impacts to the overall system quality, to associate specific transformations with the modification of the selected software features, and finally, to use such models and transformations to guide the re-engineering process. For this study, we focus on two quality requirements namely, reusability and maintainability, and on transformations that aim to migrate a procedural system to an object oriented platform. In this context, we have built prototype soft-goal graphs that relate to reusability and maintainability.

3.1. Reusability Soft-Goal Dependency Graph

Reusability aims for the design and development of software entities such as modules and classes that can be reused in different contexts without a significant effort in their adaptation.

Having as an objective to obtain through reengineering highly reusable object oriented code, we have identified a number of metrics that relate to reusability. Consequently, we have identified source code features that relate to these metrics and pertain to high modularity, low complexity and good documentation characteristics (Figure 1). These are discussed in more detail below.

High Modularity
High Modularity has long been considered as a feature that contributes towards reusability [8]. It has been argued in the software re-engineering literature that a software component is more reusable when it is highly modular, and when it provides distinct functionality. On the other hand, software components may need to be modified over their operational lifetime in order to be reused in other contexts.

We have identified the following factors that may positively affect high modularity. This is not an exhaustive list but it provides a basic framework to illustrate our re-engineering approach.

- *Encapsulation:* aims to shield data and functionality specific to a module from unauthorized access by other client modules. This can be achieved by eliminating the use of global variables, public attributes, and global flows.
- *Cohesion:* relates to the amount of functionality delivered by a software component. The more detailed the functionality delivered, the higher the cohesion level, and the lesser the modification effort to adapt a component to a new context. Cohesion can be measured by a set of metrics such as the information flow metric, and the Lack Of Cohesion in Method (LOCM) metric [4].
- *Abstraction:* relates to the use of abstract data types, abstract classes, inheritance, and polymorphism. All these design decisions aim to achieve generalization, and consequently enhance reusability.
- *Coupling:* aims to minimize inter-module dependencies, such as information flows between modules, and redundant associations, aggregations, method invocations, and abstract data type references.
- *Narrow Interface:* relates to the complexity of interface elements such as public methods, formal parameters return types, as well as modified public variables and data members. Keeping the interfaces between modules simple, the result is to limit the number of interactions and side effects a module can have, and consequently to contribute to higher understandability and modifiability.

Low Complexity
Some of the code features that may influence system complexity are:

- *Component size:* relates to the lines of code (LOC) or to the effective lines of code (ELOC). In this context, small source code size relates to low complexity and therefore leads to high reusability.
- *Component nesting level:* refers to the class inheritance depth, and the nesting level of aggregate data structure definitions. It has been argued in the software engineering literature, that the deeper the component nesting level is, the more difficult it is to achieve reuse at higher levels of the hierarchy.
- *Method complexity:* is fundamental to reducing overall program complexity and enhancing reusability. There are two ways to quantify method complexity; information flow and internal control structure. Information flow relates to complexity as measured by the number and types of formal parameters, as well as the number of method invocations. The more control and data flows a method has, the harder it is to be modified and consequently the harder it is to be reused. Similarly, the internal control structure of a component relates to the complexity of the control flow graph and it is measured by the McCabe complexity and the Knot count metric.
- *Class level complexity*: is specific to object oriented systems, but it can be measured in the similar way as it is measured at the module level in procedural systems. In this context, low class level complexity implies high adaptability and therefore high reusability.

Documentation
Although specification documents, design documents, and user manuals can facilitate reuse, for our work we mostly focus on the source code based features. In this context, we consider the consistent use of informal information, and the ratio of commented lines of code to the total size of the source code as factors that may contribute to analyzability, modifiability, and reusability.

3.2. Maintainability Dependency Graph

Software maintenance is considered as one of the most costly phases of the software life cycle. Over the past years, significant efforts have been devoted to devise techniques to minimize software maintenance costs [5].

According to the definition provided by in the ISO/IEC 9126: 1991(E) standard, maintainability is further divided into four qualities namely, analyzability, changeability, stability and testability. The subsections below discuss some of the source code features that relate to maintainability. These features do not aim to provide a complete list but to merely illustrate their use in the proposed approach.

High Analyzability
High analyzability facilitates the isolation of defects or causes of failures, and aims to identify parts of the system that need be modified for maintenance purposes (adaptive, corrective, perfective) [18]. Some of the identified features that relate to high analyzability are listed below.

- *Simple code structure*: high modularity and low complexity can result from simple code structure. High modularity makes the code structure clear and lessens the effort to comprehend it. As it has been mentioned above, source code complexity relates to a number of source features such as source code size, method level complexity, and class level complexity. It is worth noting that polymorphism facilitates modifiability and therefore maintainability [6]. Conversely, excessive class nesting levels lead to the difficulty in understanding the source code and therefore affecting negatively system maintainability [7].

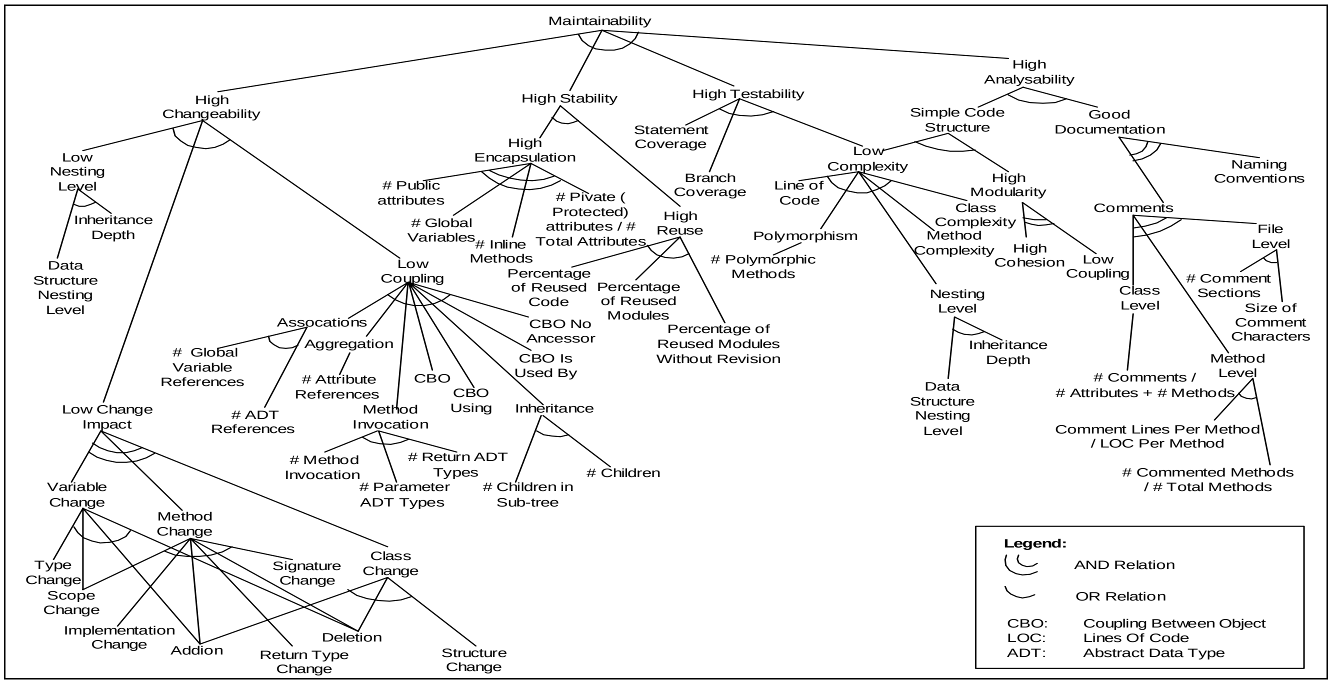


Figure 2: Maintainability Soft-Goal Dependency Graph

High Changeability
Changeability refers to the ease of modifying a system in order to remove defects, enhance its functionality, or adapt it to new platforms [18]. The following source code attributes have been identified as features that may to contribute towards increasing changeability.

- *Low nesting level:* lessens the inter class dependencies and eases the constraints to modify components.
- *Low coupling:* eases the maintenance efforts due to limited dependencies that may occur in a system.
- *Low change impact:* refers to the property of ripple effect that occurs when the state of a variable or an object changes. The analysis of such effects can be performed using data flow analysis, design documents, and informal information analysis [8]. The lower the ripple effect, the higher the maintainability. Figure 2 illustrates source code features that may lead to low change impact.

High Stability
Stability refers to the conformance of the system with respect to its specification in the case of unexpected operating conditions [18]. For this work, we have identified two factors to support stability.

- *High encapsulation:* aims to shield all the essential internal characteristics of a component (i.e. data elements, implementation details) from other external modules. It has been argued in the software engineering literature that a separation of the contractual interface from the rest of the implementation improves system stability and therefore modifiability.
- *High reuse:* refers to libraries and inheritance that are key factors to improve testability, and reduce the number of faults over time. Consequently, reuse aims for software that is more stable than systems developed by the utilization of newly developed components.

High Testability
Testability refers to the effort required for validating a software system [18]. Among other properties, the code that has no jump statements or no excessive use of decision statements is inherently simpler than software that has complex data and control flows and therefore requires less effort to test [7].

3.3. 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. The sections below discuss in more detail such as quality driven migration framework.

4. Quality Driven Software Transformations

As it has been discussed in the previous section the soft-goal dependency graphs systematically model source code features that are related to a software quality. Moreover, soft-goal dependency graphs provide a guideline on how to measure desired system qualities. 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. reusability). The objective thus is to identify the optimal combination of transformations that may yield the specified quality requirements for the target migrant system. For this work, we adopt an approach that is based on Markov models and the Viterbi algorithm to identify the transformations and the intermediate states that optimize the selected features and therefore have the highest potential of achieving the specific target qualities and requirements.

4.1. Markov Models

Markov models are directed graphs where the transitions are labeled by probability scores. Figure 3 illustrates such a model with four states and six transitions. The gray nodes denote entry and exit points. The initial state is S_0 and the final state is S_3. The arcs represent the transitions from one state to another, and are labeled by a probability value. Markov models allow for the abstraction of a continuous and complex process into a more computable form.

As illustrated in the Figure 3, the process can proceed through different paths to reach the end state. These include the paths:

$S_0\ S_1\ S_3$,
$S_0\ S_2\ S_3$,
$S_0\ S_1\ S_1\ S_3$, or
$S_0\ S_2\ S_2\ S_3$.

Each state can be further qualified by a score value. The different scores of reaching the end state from the initial state for our example are:

$S_0\ S_1\ S_3 = 0.6 \times 0.6 = 0.36$,
$S_0\ S_2\ S_3 = 0.4 \times 0.4 = 0.16$,
$S_0\ S_1\ S_1\ S_3 = 0.6 \times 0.4 \times 0.6 = 0.144$, and
$S_0\ S_2\ S_2\ S_3 = 0.4 \times 0.6 \times 0.4 = 0.096$.

In this example, the best path is the one that entails the sequence $S_0\ S_2\ S_3$.

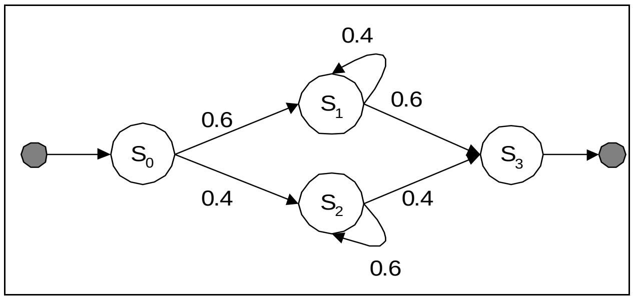


Figure 3: A Markov Model Example

4.2. Migration Process Model

Conceptually, the software migration process can be modeled as a sequence of transformations, t_{01}, t_{02}, …, t_{ij}, $t_{i,j+1}$, …, t_{kn}, and a sequence of states, s_0, s_1, …, s_i, s_{i+1}, …, s_n, where t_{ij} is the transformation applied to state s_i yielding state s_j, with a likelihood score p_{ij}. Each state, s_i, is qualified by a set of source code features chosen from the soft-goal dependency graphs and represents the outcome of the system at the transformation step τ. The transformations t_{ij} aim to transform in a stepwise fashion a legacy component written in a procedural way to a new object oriented platform. The hypothesis is that each transform, t_{ij}, causes changes to state s_i and yields system state s_j. Based on the selected source code features and the software qualities of the state, s_i and s_j, as measured by the corresponding metrics, we aim to quantify that the transformation, t_{ij} can achieve the quality goals with likelihood p_{ij}. Therefore, each of the transformations, t_{ij} is associated with a likelihood score, p_{ij}. The likelihood scores for all states can be represented by a matrix, as shown in the equation (1).

$$A = \begin{pmatrix} p_{00} & p_{01} & \cdots & p_{0n} \\ p_{10} & p_{11} & \cdots & p_{1n} \\ \cdots & \cdots & \cdots & \cdots \\ p_{n0} & p_{n1} & \cdots & p_{nn} \end{pmatrix} \quad (1)$$

The following subsections discuss the quality measurements and the calculation of the optimal path using the Viterbi algorithm.

4.3. Transition Scores

Before we discuss the computation of the likelihood scores associated with a transition, we define the following transformation 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 discussed in Section 3.

Rule 3: Two states, s_i and s_j must be distinct with respect to the identified source code features, as shown Figures 1 and 2. The examples of the selected source code features are illustrated in Figure 4.

The objective is to identify the optimal sequence of such transformations that bring the system from its initial procedural state s_0 to its final object oriented state s_n,. in a way that the source code features that affect the desired qualities are optimized.

For the state s_i, the values of these features can be represented as a vector, $<a_1, a_2, \ldots, a_k, \ldots, a_m>$, where a_k quantifies a source code feature in a numeric format. As stated in Rule 1, a transformation makes changes to states. 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. The following formula (2) is proposed to evaluate the likelihood $p_{(G)ij}$, that the transformation t_{ij} improves the quality characteristics of the system with respect to the quality goal *G*.

$$\mathrm{p}_{(G)ij} = \frac{\sum \text{Positve Impact} - \sum \text{Negative Impact}}{\sum \text{Attribute}} \qquad (2)$$

As identified in Figure 1, the high abstraction is examined by the count of the following features.

a_1:	number of references to abstract classes,
a_2:	ratio of inherited methods,
a_3:	depth of inherited children,
a_4:	number of children,
a_5:	number of overridden methods,
a_6:	number of parents, and
a_7:	ratio of inherited attributes.

For example if s_i is denoted by the attribute vector v_i = <3, 0.4, 1, 3, 2, 1, 0.2> and after t_{ij}, s_j is denoted by the attribute vector v_j = <4, 0.5, 1, 2, 3, 1, 0.2> the $p_{(G)ij}$ is equal to 2/7 since 3 out of 7 features have been increased positively for the specific quality but 1 out of 7 has been decreased.

In some cases that the negative impacts are larger than positive changes, we take the logarithm of the result. Therefore, the formula (2) is modified as following.

$$\mathrm{p}_{(G)ij} = \frac{e^{\frac{\sum \text{Positve Impact} - \sum \text{Negative Impact}}{\sum \text{Attribute}}}}{1 + e^{\frac{\sum \text{Positve Impact} - \sum \text{Negative Impact}}{\sum \text{Attribute}}}} \qquad (3)$$

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 (4). 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.

$$p_{ij} = \frac{e^{\sum_{k=1}^{R} c_k p_{(k)ij}}}{1 + e^{\sum_{k=1}^{R} c_k p_{(k)ij}}} \qquad (4)$$

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

The above formula (4) 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 $p_{(G)ij}$, only depends upon the immediately preceding states s_i, and not upon other previous states.

4.4. Optimal Transformation Path

Based on the Markov Model 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 [9] is used. The algorithm is based on dynamic programming and computes the optimal path among all the possible ones. In the Viterbi algorithm, $\delta_j(\tau)$ is defined as the highest likelihood score at step τ along a single path that ends in state s_j. The score can be recursively calculated with the formula (5) given below:

$$\delta_j(\tau) = \max_{1 \le j \le N}(\delta_i(\tau - 1) \cdot p_{ij}) \qquad (5)$$

The cumulative result of each step is stored in the vector $\varphi_j(\tau)$. This vector $\varphi_j(\tau)$, defined below in (6) and is simply a pointer to the best preceding state s_i. More details on the applications of the Markov Models and the Viterbi algorithm can be found in [9].

$$\varphi_j(\tau) = \arg\max_{1 \le j \le N}(\delta_j(\tau - 1) \cdot p_{ij}) \quad (6)$$

5. Experiments

To investigate the correctness of such a quality driven re-engineering framework, we apply it for the migration of the *gnu* AVL tree library from its original procedural implementation to an object oriented one. The *gnu* AVL library is a medium size public domain system composed of 4KLOC of C code.

The experiment applies the transformations proposed in [25]. These include:

- The transformation of global variables to classes,
- The transformation of aggregate types appearing in formal parameter lists to classes,
- The transformation of functions to methods,
- The attachment of methods that could be attached to more than one classes according to an evidence model presented in [25] and
- The transformation of the initial object model to a new one that uses polymorphism and inheritance.

5.1. Quality Goals and Metric Collection

For this case study, the target requirements for the new system were to achieve high encapsulation, high abstraction, as well as high cohesion and low coupling. These quality attributes can be considered as sub-goals, and consequently achieve higher-level goals such as, reusability and maintainability. For each of the sub-goals, a set of metrics was considered, according to the soft-goal dependency graphs shown in Figure 1 and 2. The attribute vectors for these sub-goals are illustrated in Figure 4.

5.2. Transformations and State Evolutions

Briefly, the objectification process can be separated into three sets of transformations. The first transformation set aims to achieve high encapsulation where potential classes are identified from the procedural code. The potential classes and their corresponding data members are generated from the data members of user defined types and aggregate types which refer either to global variables, or variables appearing in formal parameter lists, or to local variables within the scope of a function. Similarly, methods are attached to classes based on an evidence model that considers parameter types, return types and global variable uses [25]. The initial result of the identified classes is illustrated in Figure 5.

Encapsulation **<NPA, NGV, PAR>**, where

NPA: Number of Public Attribute **NGV:** Number of Global Variable
PAR: Private Attributes Ratio

Abstraction **<MIR, NPM, DIC, NC, NOM, NP, AIR>**, where

MIR: Method Inherited Ratio **NPM:** Number of Polymorphic Method
DIC: Depth of Inherited Children **NC:** Number of Children
NOM: Number of Overriden Methods **NP:** Number of Parents
AIR: Attribute Inherited Ratio

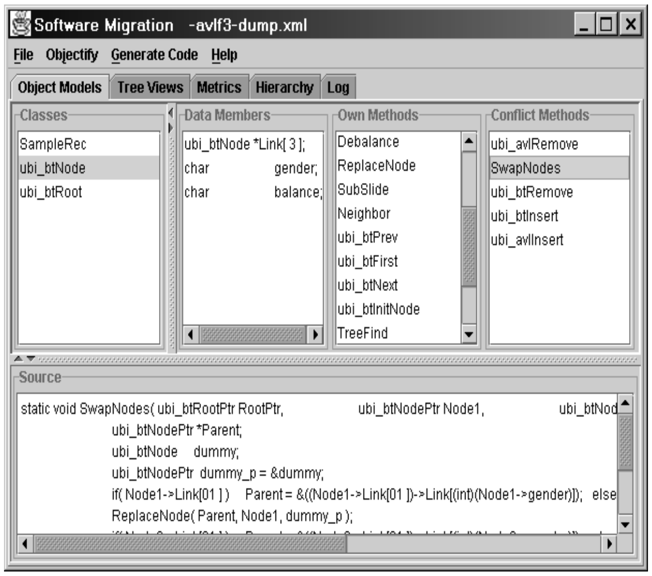
Cohesion **<IFIC>**, where

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


Software Migration -avlf3-dump.xml
File Objectify Generate Code Help
Object Models Tree Views Metrics Hierarchy Log
Classes
Data Members
Own Methods
Conflict Methods
Source

```
SampleRec
ubi_btNode
ubi_btRoot
```

```
ubi_btNode *Link[ 3 ];
char            gender;
char            balance;
```

```
Debalance
ReplaceNode
SubSlide
Neighbor
ubi_btPrev
ubi_btFirst
ubi_btNext
ubi_btInitNode
TreeFind
```

```
ubi_avlRemove
SwapNodes
ubi_btRemove
ubi_btInsert
ubi_avlInsert
```

```
static void SwapNodes( ubi_btRootPtr RootPtr,          ubi_btNodePtr Node1,          ubi_btNod
              ubi_btNodePtr *Parent;
              ubi_btNode    dummy;
              ubi_btNodePtr  dummy_p = &dummy;
              if( Node1->Link[01 ] )    Parent = &((Node1->Link[01 ])->Link[(int)(Node1->gender)]);  else
              ReplaceNode( Parent, Node1, dummy_p );
```

Figure 5: System State with Initial Classes

Specifically, Figure 5 illustrates the initial breakdown of the system in three classes: `SampleRec`, `ubi_btNode` and `ubi_btRoot`. The right most column lists the potential methods that can be attached to more than one class. The second set of transformations aims to attach methods to classes. Specifically, when a transformation can attach a method to different classes; we say that the methods are in conflict. For example, `swapNode` is one of the methods that are in conflict, and can be assigned to either `ubi_btNode` or `ubi_btNode`. Thus, two states can be generated from the application of the specific transformation. In this case, the choice of the states is based the probability that the resulting state contributes more towards achieving the desired quality characteristics for the migrant system (i.e.

Assigned Class	ubi_btNode	ubi_btRoot
CBO	-3	-15
IFBC	0	-9
DCC	0	0
NMI	0	-3
NLVT	0	-3
NMPT	0	-9
NMRT	0	0
$p_{(coupling)ij}$ (Formula 2)	-0.14285	-0.71429
$p_{(coupling)ij}$ (Formula 3)	0.4643	0.3286

Table 1: Coupling measurement for resolving the attachment of method swapNode to a class

Assigned Class	ubi_btNode	ubi_btRoot
IFIC	+9	0
$p_{(cohesion)ij}$ (Formula 2)	1	0
$p_{(cohesion)ij}$ (Formula 3)	0.7311	0.5

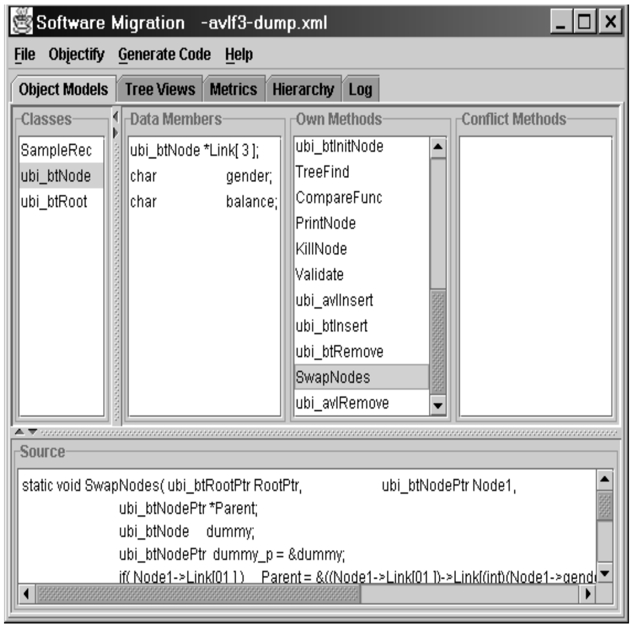
Table 2: Cohesion measurement for resolving the attachment of method swapNode to a class

Assigned Class	ubi_btNode	ubi_btRoot
p_{ij} (Formula 4)	0.6451	0.6021

Table 3: Accumulative result for resolving the attachment of method swapNode to a class

high cohesion for a class and low coupling between classes). Table 1 illustrates the changes of the features related to coupling, if the method `swapNode` 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 2 and 3, the cases of $p_{(coupling)ij}$ are calculated, respectively. Similarly, table 2 illustrates the impact on cohesion. Finally by utilizing formula (4) the accumulative result of the impact on both goals is calculated, and is shown in Table 3. Thus, the `swapNode` is assigned to `ubi_btNode`, because it has higher likelihood to achieve the desired software goals. The rest of the 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.

In the final third set of transformations, inheritance and polymorphic methods are identified applying the



Classes: SampleRec, ubi_btNode, ubi_btRoot

Data Members:

```
ubi_btNode *Link[ 3 ];
char          gender;
char          balance;
```

Own Methods: ubi_btInitNode, TreeFind, CompareFunc, PrintNode, KillNode, Validate, ubi_avlInsert, ubi_btInsert, ubi_btRemove, SwapNodes, ubi_avlRemove

Source:

```
static void SwapNodes( ubi_btRootPtr RootPtr,          ubi_btNodePtr Node1,
               ubi_btNodePtr *Parent;
               ubi_btNode    dummy;
               ubi_btNodePtr  dummy_p = &dummy;
               if( Node1->Link[01 1)    Parent = &((Node1->Link[01 1)->Link[(int)(Node1->gend
```

Figure 6: System State without Methods in Conflicts

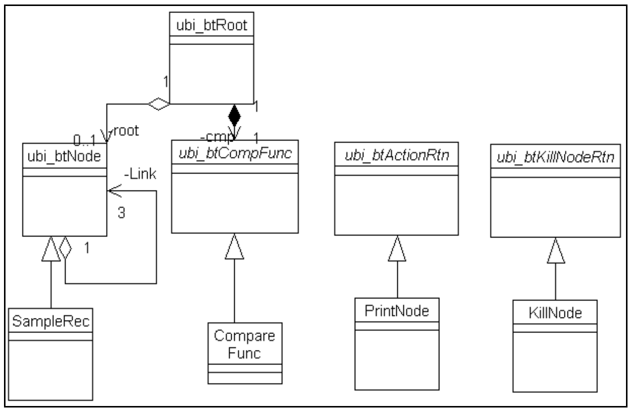


Figure 7: Final System State

proposed framework in the same way as above. The final state of the system is illustrated in Figure 7 as a UML diagram. In this transformation set, the abstraction goal is considered.

As a summary, after the application of each transformation, the impact on the quality goals is measured. At each step of the transformation process, the states with the highest likelihood towards achieving the quality objectives are selected to apply the next transformation. Although this case study presents only transformations related to achieving high abstraction, high encapsulation, high cohesion, and low coupling, hierarchical models can be considered as well. In such hierarchical models, each system state can be further modeled as a collection of sub-states and sub-transformations.

6. Conclusion

Quality is a critical issue in the process of software migration. The assumption for this work is that software migration is achieved by applying a sequence of transformations. Each transformation can move the system from one state into another state. The complete process starts with the original system to be reengineered, denoted by an initial state, and ends in a new system, denoted by a final state. Each transformation corresponds to a state transition, and is quantified by the likelihood that the resulting state is closer towards achieving the desired final quality goals. Specifically, in this paper, we propose a re-engineering framework for procedural to object oriented platforms that models the dependencies between source code features and specific software qualities, and a quantitative method that indicates the likelihood for each transformation achieving specific quality goals. By the use of the Markov models and the Viterbi algorithm, the process aims to identify the optimal set of transformations that can be applied in order to yield a target migrant system that possesses specific desired quality characteristics.

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] Aniello Cimitile, et.al, “Identifying Objects In Legacy Systesm Using Design Metrics”, The Journal of Systems and Software 44 (1999), Elsevier.

[2] L.H. Etzkorn, W.E. Hughes Jr., and C.G. Davis, “Automated reusability quality analysis of OO legacy software”, Information and Software Technology 43 (2001), Elsevier.

[3] Sen-Tarng Lai and Chien-Chiao Yang, “A Software Metric Combination Model for Software Reuse”.

[4] Shyam R. chidamber and Chris F. Kemerer, “A Metrics Suite for Object Oriented Design”, IEEE Transactions on Software Engineering, Vol 20, No. 6, June 1994.

[5] M. Ajmal Chaumun, et. al, “Design Peroperies and Object Oriented Software Changeability”.

[6] David P. Tegarden, and Steven D. Sheetz, “Effectiveness of Traditional Software Metrics for Object Oriented Systems”, IEEE 1992.

[7] Sen-Tarng Lai, Chien-Chiao Yang, “A Software Metric Combinatin Model for Software Reuse”.

[8] Lionel C. Briand, Christian Bunse, and John W. Daly, “ A Controlled Experiment for Evaluating Quality Guidenlines on the Maintainability of Object Oriented Designs”, IEEE Transactions on Software Engineering, Vol 27, No. 6, June 2001.

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

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

[16] 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.

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

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

[19] Stephen H. Han, “Metrics and Models in Software Quality Engineering”, Addison-Wesley, 1995.

[20] S.R. Chidamber, C.F. Kemerer, “A Metrics Suite for Object oriented Design”, IEEE Transaction, Software Engineering, 1994.

[21] W. Li, and S. Henry, “Object Oriented Metrics Which Predict Maintainability”, Journal of Systems Software, 1993.

[22] M. Lorenz and J. Kidd, “Object-Oriented Software Metrics”, PTR Prentice-Hall, Englewood Ciffs, New Jersey, 1994.

[23] Lionel Briand, et. al, “Characterizing and Accessing a Large-Scale Software Maintenance Organization”, http://www.cs.umd.edu/projects/SoftEng/ESEG/papers/CS-TR-3354.pdf

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

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

[26] 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, 2002.