Incremental Quality Driven Software Migration to Object Oriented Systems

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

Abstract

In the context of software maintenance, legacy software systems are continuously re-engineered in order to correct errors, provide new functionality, or port them into modern platforms. However, software re-engineering activities should not occur in a vacuum, and it is important to incorporate non-functional requirements as part of the re-engineering process. This paper presents an incremental reengineering framework that allows for quality requirements to be modeled as soft-goals, and transformations to be applied selectively towards achieving specific quality requirements for the target system. Moreover, to deal with large software systems, a system can be decomposed into a collection of smaller clusters. The reengineering framework can be applied incrementally to each of these clusters and results are assembled to produce the final system. Using the theory presented in this paper, we developed a reengineering toolkit to automatically migrate procedural systems to object oriented platforms. We used the toolkit to migrate a number of open source systems including Apache, Bash and Clips. The obtained results demonstrate the effectiveness, usefulness, and scalability of our proposed incremental quality driven migration framework.

1. Introduction

Software systems are continuously being evolved soon after their first version is delivered to meet the changing requirements of their users. Software practitioners are performing changes daily to source code such as correcting errors, adding new functionality, and adopting new technologies to ensure the users' needs are met. Otherwise the software system will be abandoned. Unfortunately, as pointed out by Lehman's laws of evolution, the quality of an evolving program tends to decline, and its structure becomes more complex [1]. Therefore, legacy systems become harder to understand and maintain after many years of evolution. To extend the lifecycle of a legacy system, migration of procedural systems into object oriented platforms provides a promising solution to leverage business values entailed in such systems. However, most legacy systems are written in a variety of procedural languages, are composed of millions of lines of code, and have deteriorating quality. It is a challenging task to devise a tractable methodology where source code transformations can be associated with specific quality improvements and can be applied for the migration of procedural systems to the object oriented platforms.

In our previous work [2], we introduced a unified domain model for a variety of procedural languages such as C, Pascal, COBOL and Fortran. This unified model is implemented in XML and denotes common features among these languages such as routines, subroutines, function, procedures, types, just to name a few. Using this unified model, we can apply standardized transformations upon systems written in a variety of procedural languages. These transformations can migrate procedural code to object oriented platforms.

Moreover, we proposed a quantitative framework [3] that monitors and evaluates software qualities at each step of the re-engineering process. This approach allows for quality requirements, such as high maintainability and reusability, to be modeled as soft-goals. To operationalize these quality requirements into the migration process, we construct the migration process as a state transition system, which consists of a sequence of transformations each one of which alters the state of the system being migrated.

The Viterbi algorithm and Markov chain type of models can identify the optimal sequence of transformations that could achieve the highest quality in the migrated system. In this respect, the re-engineering process is fine-tuned so that the migrated system conforms to specific target objectives/requirements, such as better performance or higher maintainability.

In this paper, we are particularly interested in improving the scalability of our quality driven migration framework. To keep the complexity and the risk of the migration of large system into manageable levels, we propose an incremental approach that allows for the decomposition of legacy systems into smaller manageable units (clusters). A state transition approach is applied to each such cluster to identify an object model with the highest quality. Then an incremental merging process allows for the amalgamation of the different partial object models into an aggregate composite model for the whole system. Furthermore, to avoid the state explosion for large legacy systems, we enhance the Viterbi algorithm to limit the number of states generated. In this way, the proposed quality driven migration framework can tackle large systems in an acceptable hardware and software resource requirements. Moreover, we propose techniques to validate the quality of the object models derived through our migration framework.

This paper is organized as follows. Section 2 gives an overview of the proposed quality driven software migration framework. Section 3 presents system segmentation and amalgamation processes to enable incremental migration of procedural systems into object oriented platforms. Section 4 presents cases studies that utilized the proposed approach. Section 5 discusses the related work. Section 6 concludes the paper.

2. Quality Driven Software Migration Framework

The object oriented migration process involves the analysis of the Abstract Syntax Tree (AST) of the procedural code, the identification of object models, and the generation of object oriented code. One of the major objectives of our research is to incorporate quality as an integral part of the migration process. We propose a quality driven software migration framework to achieve this objective. The focal points of the framework include: 1) the identification and the modeling of quality requirements in the target system and 2) the operationalization of these requirements into the reengineering process to produce the highest quality system. In the following subsections, we discuss these two focal points in more details.

2.1 Modeling Quality Requirements

To drive the migration process to meet specific quality requirements (such as high maintainability), we provide a software quality modeling process that elicits quality goals, models these quality goals in a measurable level, and evaluates the achievement of these desired qualities in the final migrated system. Typically, quality requirements are modeled in a top down manner, which involves identifying a set of high-level quality goals, such as maintainability, subdividing and refining these goals into more specific low level attributes (e.g., design decisions, or software code features). The lowest level attributes can be directly measurable using software metrics. In our previous work, we adopted soft-goal interdependency graphs [4] to associate source code features with high-level quality goals. In such graphs, nodes represent design decisions, and edges denote positive or negative dependencies towards a specific requirement. The leaves of such graphs correspond to measurable source code features that impact other nodes to which they are connected. A set of metrics is selected to compute the corresponding source code features, appearing as leaves in the soft-goal interdependency graph. The metric results of the leaf nodes indirectly reflect the satisfaction of their direct or indirect parent nodes in the soft-goal interdependency graph.

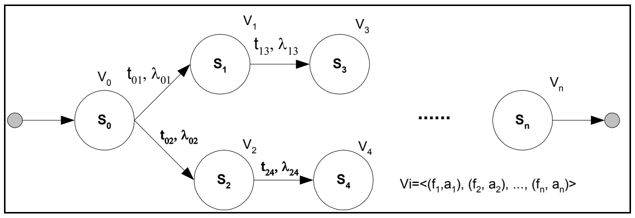

Figure 1. Quality Driven Software Migration Framework

2.2 Operationalization of Quality Requirements

The migration process can be constructed as a sequence of transformations, as depicted in Figure 1. Each transformation alters the state of the software system by converting a data type to a class candidate, associating a method to a class candidate, and defining inheritance associations. These transformations are repeated till the final object oriented system is obtained. Each transformation affects the features, appearing as leaves in the quality soft-goal graph. Consequently, we consider the transformations have a positive impact on the modeled quality. The objective thus is to identify the optimal combination of transformations that yields the best quality requirements for the target migrant system. We define the migration process as follows.

Definition 1: A quality driven migration process is a tuple **< S, I, F, T, G, V, A, $\xrightarrow{a}$ >.**

- **S** is a set of non-empty states, s_0, s_1, ..., s_i, s_{i+1}, ..., s_n. State, s_0, represents the original software system. Other than the initial state, the set **S** can contain more than one final states that correspond to different resulting migrant systems (i.e. alternative object models), or to an empty state that denotes failure. The states between the initial state and final states represent a snapshot of the migration process whereby a mix of procedural model and object oriented model exist as the system evolves from its original procedural state to a fully object oriented final state.
- **I** represents the initial state s_0 .

- **F** represents a set of final states that correspond to different object models for the initial procedural system.
- **T** is a catalogue 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. These transformations aim to convert a software system in a stepwise fashion from its initial state (original procedural system) to a final state (new object oriented system). t_{ij} represents a transformation moving from s_i to s_j .
- **G** is a set of soft-goal interdependency graphs that elaborate high-level non-functional requirements into measurable source code features.
- **V** is a set of feature vectors, v_0, v_1, ..., v_i, ..., v_n, in which v_i is a feature vector for s_i and represented as $<(f_1, a_1), (f_2, a_2), ..., (f_k, a_k), ..., (f_m, a_m)>$, where f_k is a terminal feature in the soft-goal interdependency graphs, and a_k the metric value of this feature in state s_i.
- **A** is a set of actions, a_{00}, a_{01}, ..., a_{ij}, $a_{i,j+1}$, ..., a_{kn}, which is a pair of a transformation, t_{ij}, and its quality likelihood, λ_{ij}, which indicates that a transformation contributes towards a desired quality goal.
- $\xrightarrow{a} \subset S \times A \times S$ is a set of transformation rules, which define the semantic meaning and constraints for each transformation. Transformations in **T** are selected from a transformation rule catalog [5].

As illustrated in Figure 2, a full transformation path can be formed by the composition of transformations from the original state to the final states. For example, we can identify three possible class candidates from the original system, s_0. This leads to the generation of multiple states (e.g., s_1, s_2, and s_3) of which each state contains one possible class candidate. Transformations can be further applied to class candidates in each state, such as assigning methods to class candidates. A consecutive state can be generated to reflect the method assignment to the identified class candidate. Conflicts might occur where one method might be assigned into more than one class candidates. This causes the generation of multiple states of which each state is produced by assigning the conflicting method to an alternative class. Therefore, multiple transformation paths are generated. Each path represents an alternative object model. The migration process continues until no further transformations can be applied. Our objective is to determine the optimal path that achieves the desired quality requirements, such as high maintainability and good reusability. Our previous work provides quantitative methods to quantify the magnitude by which applying a transformation may contribute towards achieving a desired system quality [3].

To summarize the idea, each transformation is labeled

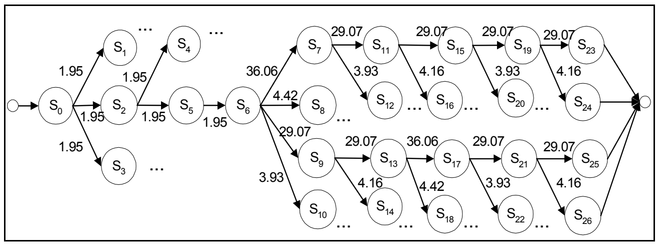

Figure 2. An Example Transformation Chain

with a quality likelihood, λ_{ij}. The likelihood of a transformation path is calculated by multiplying the likelihoods of the transformations along the path. All the transformation paths are generated using the Vertibi algorithm. An optimal transformation path can be selected from the path with the highest quality likelihood. In the end, a quality validation can be performed to examine whether the optimal path generates the object model with the highest quality.

3. Incremental Migration Process

Legacy software systems are usually large systems. Our proposed approach must scale well to handle the migration of these systems. In this section, we present two enhancements to our quality driven migration approach. Firstly, we present a technique to break a large system into a set of clusters. Each cluster is migrated independently. By applying the proposed quality driven migration framework on the clusters, a partial object model can be identified from each cluster and merged incrementally into a full object model that produces a final migrated system. Secondly, we modify the Viterbi algorithm with constrains on the number of states generated, in order to reduce the search space for the optimal transformation path.

3.1 System Segmentation

To reduce the time and space requirements needed to migrate a large legacy system in one sweep, we break the legacy system into clusters. Most clustering techniques presented in the literature utilize certain criteria to decompose a system into a set of meaningful modular clusters. Such criteria attempt to achieve a cluster with low coupling, high cohesion, minimal interface or sharing. In the context of our research, our objective is to produce clusters that have the least data dependencies on other clusters. This would enable us to migrate clusters independently. Moreover, the order of selecting which cluster to migrate would have no effect on the final object model. To achieve this objective, each cluster contains the maximum number of source code entities that are related to a class candidate. A source code entity that initiates the formation of a cluster is called a seed. Other entities that associate with this seed entity are used to form a cluster.

An association is a directed edge from a seed to its related entities.

A seed, refereed as, T_i, is selected according to its potential to be considered as a class candidate in the final migrated system. In this context, a seed entity can be chosen from aggregate data types, global variable declarations, and function pointer declarations. Specifically, the aggregate data types include *struct* type definitions, *union* type definitions, *arrays*, and *enumeration* definitions. In this case, fields in an aggregate data type become data members in a class candidate. Similarly, a global variable is encapsulated as a data member in a class candidate. Moreover, a function pointer declaration is treated also as a seed entity for the reason that a function pointer declaration defines a type for the functions passed as parameters.

Due to the object oriented design principle that a class encapsulates data and related methods, we focus on the discovery of relations between the seed and other data types and functions that use the seed. These relations can be type references, data updates, or data uses. Specifically, a seed type can be referred by functions and data fields of other data types. The entities associated with the seed are represented as a set of data type declarations, $\{T_j\}$, and a set of functions $\{F_k\}$. We define a cluster as tuple, described in Definition 2.

Definition 2: A cluster is represented as a tuple, $< T_i, M_{T_i}, R_{T_i} >$.

- T_i, the seed of a cluster, has a potential to become a candidate class
- M_{T_i} is a set of functions that use or update the seed T_i
- R_{T_i} is a set of data types that have data fields refereed to the seed T_i

The system decomposition process is conducted in the following steps:

1. Identify the possible seeds;
2. Associate the related entities with each seed;
3. Generate a set of clusters, denoted as a set of tuples, $\{< T_i, M_{T_i}, R_{T_i} >_j\}$.

Figure 3 illustrates a result of system segmentation, where three clusters have been identified:

$C_0 = <T_0, \{F_0, F_1\}, \{T_1\}>$,
$C_1 = <T_1, \{F_2, F_4\}, \varnothing>$,
$C_2 = <T_2, \{F_3, F_4\}, \{T_1\}>$.

As illustrated in this example, the system segmentation process allows overlapping entities between clusters. When an overlap occurs on aggregate data types, it may indicate a multiple inheritance relationship among the generated class candidates from each cluster. If an overlap occurs on functions it reflects conflicts in methods. In this case, multiple possible transformation paths can be generated by assigning the methods into alternative class candidates. The migration process framework (described in Section 2) provides a qualitative method to determine the class candidate to attach the function. Finally, to independently select and migrate a cluster without relying on the information in other clusters, shared entities among clusters are duplicated in each cluster. Some functions may not be related to any seeds, these are grouped into a "leftover" cluster.

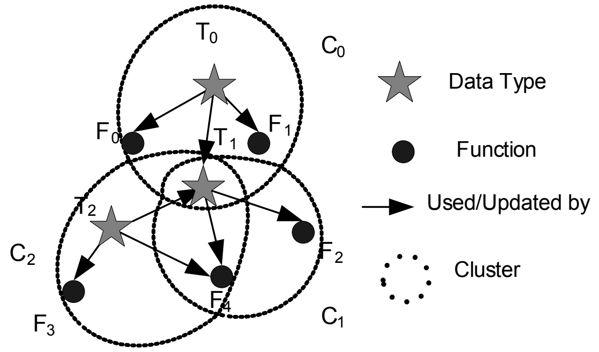

Figure 3. Example for System Segmentation

3.2 System Amalgamation

The system segmentation process decomposes the program into a set of smaller clusters, which contain the ASTs of segments of the procedural code to be migrated. In this respect, the migration process is divided into phases, as depicted in Figure 4. Each phase transforms one cluster into an object model. In particular, the initial object model OM_0 is empty in the first phase. After the transformation of Cluster 1, C_1, the first object model OM_1 is generated and serves as input to the second phase, along with a new cluster. Consequently, after each phase a new intermediate Object Model, OM_k, is generated and serves as an input to the following reengineering phase. In this sense, the result from the present phase and the preceding phase are merged, and the new intermediate object model is produced.

The amalgamation process iterates over each cluster, and updates the system object model. The transformations in each cluster are performed in four steps:

1. Generate a new class which is added into the object model;
2. Assign functions from the set of associated functions to the new class candidate, and update the object model;
3. Resolve conflicts when a function can be assigned to either the newly identified class candidate or to existing class candidates in the object model;

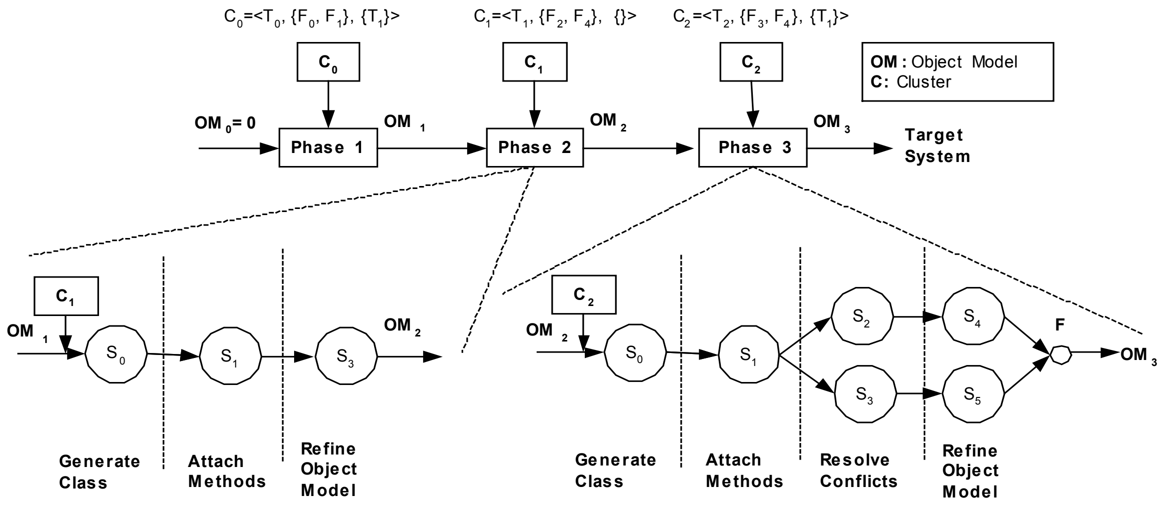

Figure 4. Amalgamation Process for Merging Object Models

Table 1. An example for Incremental Migration Process

Phase	Cluster	States	Object Model
1	$C_0 = <T_0, \{F_0, F_1\}, \{T_1\}>$	0	Class T_0: {}
		1	Class T_0: $\{F_0, F_1\}$
2	$C_1 = <T_1, \{F_2, F_4\}, \varnothing>$	0	Class T_0: $\{F_0, F_1\}$, Class T_1:{}
		1	Class T_0: $\{F_0, F_1\}$, Class T_1:$\{F_2, F_4\}$
		2	Class T_0: $\{F_0, F_1\}$, Class T_1:$\{F_2, F_4\}$
3	$C_2 = <T_2, \{F_3, F_4\}, \{T_1\}>$	0	Class T_0: $\{F_0, F_1\}$, Class T_1:$\{F_2, F_4\}$, Class T_2:{}
		1	Class T_0: $\{F_0, F_1\}$, Class T_1:$\{F_2, F_4\}$, Class T_2:$\{F_3, F_4\}$
		2	Class T_0: $\{F_0, F_1\}$, Class T_1: $\{F_2\}$, Class T_2: $\{F_3, F_4\}$
		3	Class T_0: $\{F_0, F_1\}$, Class T_1: $\{F_2, F_4\}$, Class T_2: $\{F_3, \}$
		4	Class T_0: $\{F_0, F_1\}$, Class T_1:$\{F_2\}$, Class T_2:$\{F_3, F_4\}$
		5	Class T_0: $\{F_0, F_1\}$, Class T_1:$\{F_2, F_4\}$, Class T_2:$\{F_3\}$
		F	object model, OM_3, from the transformation path with the highest likelihood

4. Refine the object model by identifying class associations such as class inheritance.

In a nutshell, the proposed quality driven migration approach is applied iteratively to every cluster. In each cluster, the migration process is divided into a sequence of states and transformations. During the migration of the first cluster, the initial state represents purely procedural code of the cluster. Once the optimal path in a cluster is identified, the final state from the optimal path in a cluster serves as the initial state for the next cluster. The object model is incrementally expanded by identifying additional classes from the new clusters. For example, the clusters in Figure 3 are used to illustrate the application of the amalgamation algorithm process using the quality driven migration framework. In this example, the amalgamation process is divided into three phases (a phase for each cluster), as depicted in 4. In each phase, one cluster is migrated by following the four steps as described above. The evolution of the object model in each state is specified in Table 1. In Phase 1, cluster $C_0 = <T_0, \{F_0, F_1\}, \{T_1\}>$ serves as an input. Two states are generated $\{s_0, s_1\}$. In s_0, T_0 is identified as a class. In s_1, two methods: F_0 and F_1 are assigned to class T_0. In Phase 2, $C_1 = <T_1, \{F_2, F_4\}, \varnothing>$ serves as input along with the object model (OM_1) from Phase 1. T_1 is identified as a new class form C_1 in s_0. In s_1, methods F_2 and F_4 are assigned to T_1. There are no methods in conflict in Phase 2. As shown in Figure 3, T_1 is a data type and one or more of its data fields refer to T_0. This indicates that an inheritance relation exists between these two types. The class inheritance is determined based on the context of the usage of T_0 and T_1 in the source code and a set of heuristics. For this example, the class inheritance identification is omitted (see [5]). In Phase 3, $C_2 = <T_2, \{F_3, F_4\}, \{T_1\}>$ serves as input to the migration process along with the object model (OM_2) from Phase 2. In s_0, T_2 is the new class identified from the C_2. In s_1, two methods, F_3 and F_4 are assigned to T_2. T_2 and T_1 have common method F_4. Therefore method F_4 is in conflict.

Two states s_2 and s_3 are generated form s_1 by assigning the conflicting method F_4 to either class T_1 or class T_2. Meanwhile, the quality features and the corresponding likelihoods for both transformations (function assignments) are computed based on the affected features. In the end, the resulting object model is selected from the transformation path with the overall highest likelihood to achieve the best quality.

The incremental process can be halted at any phase; and the intermediate object oriented system can be generated and put into operation. There are two ways to enable the newly migrated object oriented cluster and the legacy procedural code to be integrated with the rest of the existing system. 1) All of the remaining unprocessed procedural clusters could be directly converted into a single class. That is, one cluster is transformed into one class. All aggregate types and variable declarations in the cluster become public data attributes of the newly created class. The functions in a cluster become public methods of the class. In this way, the whole system is converted into an object oriented design. This brute force migrated object oriented design will likely not have as high quality as one derived using our migration framework. 2) Alternatively, the rest of the non migrated system may be considered as a one big component. By the use of wrapping techniques, the rest of the non migrated system and the newly identified object oriented system can interact using middleware technologies. Various wrapping and middleware technologies are presented in [6].

3.3 Transformation Path Generation

In our quality driven migration approach, we consider that the transformation path with the highest likelihood score to be the path most likely to produce the highest quality in a migrated system. Therefore, a main emphasis of our migration approach is to determine the optimal transformation path. We use the Viterbi algorithm to generate all the transformation paths and identify the optimal path. A draw back of using the Viterbi algorithm is the need to retain all states and paths before identifying an optimal path. In our research, we developed a number of heuristics to reduce the number of generated states and the time required to find the optimal path. These heuristics are essential to permit our approach to migrate large legacy systems with reasonable resource requirements. These heuristics are discussed below.

Transformation rule application constraints
Transformation rules provide means to implement generic migration steps. Each of the rules alters a set of source code features. We associate pre- and post-conditions with each transformation rule to efficiently specify constraints to apply transformation rules in procedural code features. In the migration process, pre-conditions are the procedural source code features that a transformation can operate on and convert them into object oriented structures, as specified in post-conditions. Therefore, the pre/post conditions indicate the order to apply the transformations on states, and place constraints on the expansion of the optimal transformation path search tree.

Hierarchical state modeling
When a segment of the search tree contains a sequence of states in a single path, these states can be collapsed into one composite state. The likelihood of the composite state is equal to the multiplication of the likelihoods of transformations along the single path inside the composite state. For example, as illustrated in Figure 4, s_0 and s_1 of Phase 3 can be merged into one state to reduce the number of states generated, because there aren't other states emanating from s_0. This merging will not affect the choice of the optimal transformation path.

Incremental object model generation
Adopting the incremental migration approach, described in Section 3.2 and 3.3, we reduced the complexity and the space of the optimal transformation path search tree. At each cluster, we concentrate on identifying one class candidate, building the search tree regarding this one class candidate, and updating the system object model incrementally one class at a time.

Implementation considerations
The implementation of the system states is crucial in reducing the complexity of the proposed state transition approach. The main idea is to simplify the representation of states and minimize the space required to retain states in memory.

Sub-optimality to achieve performance
The sub-optimality refers to the number of applied transformations and how well the desired quality objectives are met in the new migrated system. The Viterbi algorithm used to locate the optimal path requires the generation of all possible paths. This limits the applicability of our migration approach, as we may need to generate a large number of paths even for medium size systems. Alternatively, we can use the A* algorithm to locate the optimal path. In this case, not so promising transformations can be also considered with the expectation that they may later produce an optimal result. Therefore we may locate a sub-optimal optimal transformation path. We limit the application of A* only to clusters with large number of states to ensure that we achieve the optimal path for most of the clusters.

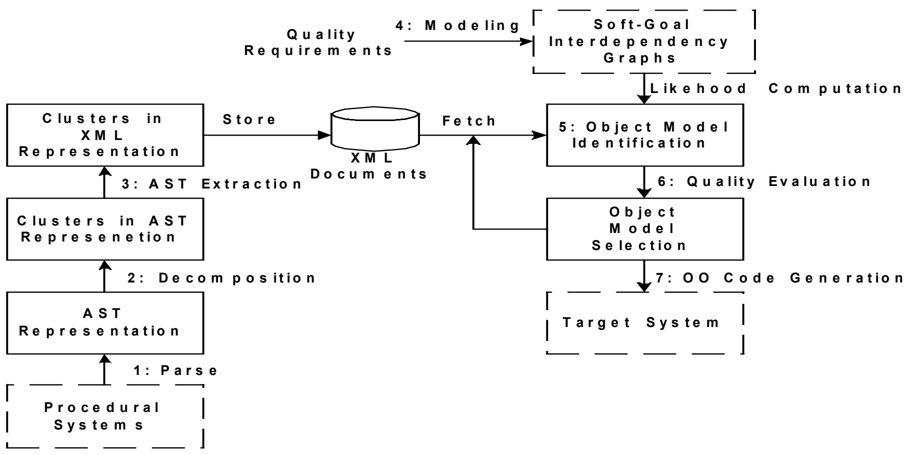

Figure 5. Incremental Migration Steps

Table 2. Characteristics of the Procedural Systems

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

4. Case Studies

To investigate the effectiveness of the incremental quality driven migration framework, a comprehensive set of case studies are conducted and presented in this section. The case studies are performed on three medium size open source procedural systems from different domains: 1) The Apache Web Sever 2) The BASH Unix Shell and 3) The Clips Expert System Builder. Table 2 illustrates the source code related characteristics of these systems. A prototype software toolkit is developed using the theoretical approach described in this paper.

In this section, we firstly give an overview of the toolkit implementation and highlight the scalability and performance issues that may arise during the migration process. Secondly, we study the efficiency of the system segmentation process. Thirdly, we evaluate complexity of the state transition approach. Finally, we examine the quality of the migrated systems, and verify that the migrated system generated from the optimal path achieves the highest quality.

4.1 Incremental Migration Steps

The toolkit implements the proposed incremental quality driven migration in seven steps, as illustrated in Figure 5:

Step 1: The original legacy systems is parsed and represented in terms of its Abstract Syntax Tree (AST).

Step 2: Using the techniques described in Section 3.1, we decompose the AST into several clusters. Each cluster intends to include extensive source code features relating to a class candidate.

Step 3: We defined a generic procedural language domain that contains the common constructs in various procedural languages. Due to space limitation we do not cover this generic procedural language domain [2]. The AST of each cluster is converted to conform to this generic procedural language domain and is represented in XML. In this way, we can use a set of generic transformations that can migrate procedural systems written in different procedural languages into functionally similar object oriented systems. The generated XML documents are placed in a central repository.

Step 4: The high-level quality requirements for the target migrant systems are elicited based on domain knowledge, customer interviews or documentation. The focal point is to refine the high-level quality requirements into measurable design decisions and source code features. The refinement process is modeled using soft-goal interdependency graphs.

Step 5: The migration process is operated iteratively on each cluster, as described in Section 3.2. For each cluster, we apply the proposed quality driven migration process. We then update the overall system object model with newly identified object model from the processed cluster.

Step 6: The quality evaluation process selects the object model with the best quality using the Viterbi algorithm and computes quality improvement likelihood scores. We validate that the target object model achieves the high cohesion and low coupling using object oriented six metrics.

Step 7: Once all the clusters are migrated. A final object model is obtained. The object oriented code is generated from the object model with the highest quality.

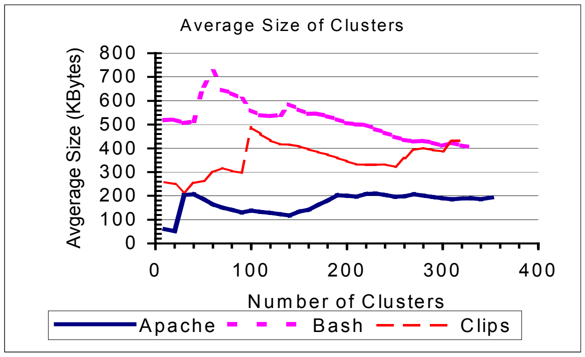

Figure 6. Average Cluster Size During the System Segmentation Phase

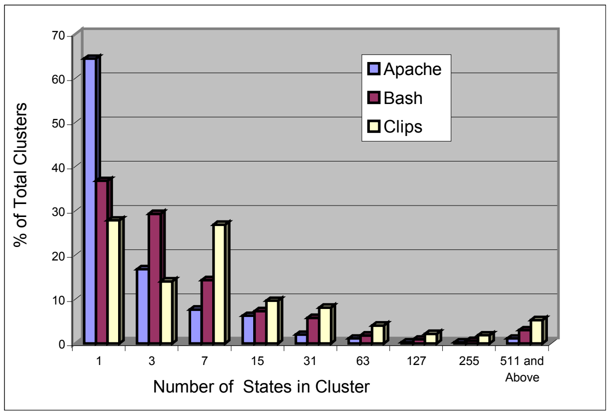

Figure 7. Distribution of Number of States in Clusters as Percentage of Total Clusters

4.2 Experiments on System Segmentation

In this section, we present experiments that measure the average size of the generated clusters using the system segmentation algorithm. The size of clusters is crucial for the success of a reengineering project due to the amount of computation resources needed. Especially, the representation of Abstract Syntax Trees in an XML format (Step 3) increases the size of the source code representation dramatically. For example, the CLIPS system (34KLOC) represented in XML requires 38MB. Furthermore, due to the possible large size of the procedural system, it may become an intractable task to migrate the entire code of a large system in one sweep. The system segmentation algorithm reduces a large procedural system into a set of manageable clusters. The average disk space for each cluster is illustrated in Figure 6. The X axis represents the number of the clusters. The Y axis refers to the average disk space of a cluster related to the corresponding number of clusters. The values of Y axis are computed using Formula 1. Formula 1 details the calculation of the average size of clusters as shown in Figure 6. The average size of a cluster varies based on the number of clusters counted. The order of clusters measured is based on the order of cluster generation during the segmentation phase. Figure 6 shows several peaks because the generation of clusters of large size increases the average size of clusters.

$$Average\ Size\ of\ Clusters = \frac{\sum_{i=1}^{i=K} SizeofCluster_i}{K} \qquad (1)$$

K=1, …, M, M is the total number of clusters

The results in Figure 6 illustrate that the average size of clusters is relatively small. For example, the average size for the Apache system is 200 Kbytes, Bash 400 Kbytes, and Clips 400 Kbytes XML clusters. Moreover, the largest size of a cluster is approximately 2MB. The size of the cluster demonstrates that the proposed framework can operate on large systems at a steady rate without extraordinary resource requirements.

4.3 Complexity of the State Transition Approach

In this section, we present the complexity of the state transition approach used to identify an optimal transformation path. Essentially, the complexity of the approach can be measured in three aspects: 1) the number of states generated in a cluster, 2) the time required to process a cluster, 3) the memory needed to represent a state. Our objective is to validate that the incremental migration process can effectively shorten the search space and improve the scalability of the proposed framework.

As discussed in Section 3, the migration is conducted iteratively one cluster at each step. A new class candidate is identified from the initial state and the overall system object model is updated. One method might be assigned into more than one class candidates. When a method assignment conflict is encountered, multiple states are generated to compute the optimal transformation path. To avoid the space explosion, it is critical to resolve conflicts in the scope of one cluster. In this context, the transformation paths are modeled by a binary tree, because the conflicting method is assigned into either class. Moreover, we utilize the heuristics discussed in Section 3.3 to compress the states. Therefore, if there is no method in conflict, only one state exists in the cluster. The number of the states is calculated by Formula (2) where i refers to the number of methods in conflict.

$$num.ofstates = \sum_{i=0}^{n} 2^i \qquad (2)$$

Figure 7 shows the distribution of states generated in the clusters. A large number of the clusters have only one state – 64% for Apache, 36% for Bash, and 27% for Clips. This is due to the state reduction heuristics used in Section 3.3. In addition, these clusters require minimal processing time, as there is no need to perform any optimal transformation path searching in these clusters.

We also examine the average processing time for each cluster. Average processing time includes building a

Table 3. Average Processing Time for Identifying Optimal Transformation Path

Number of States	Apache (ms)	Bash (ms)	Clips (ms)
3	75	155	70
7	249	230	387
15	568	348	569
31	1,582	990	701
63	2,628	1,656	930
127	8,687	3,671	3,939
255	N/A	6,453	5,278
511	41,968	14,554	4,101
1023	N/A	17,625	5,766
2047	N/A	121,562	18,515
4095	N/A	N/A	50,286
8191	N/A	123,828	2,099,672

binary tree, calculating a likelihood score for each state, and finally identifying the optimal path. Table 3 summarizes statistics of the states generated while identifying optimal transformation paths among the three examined systems. The table shows the results for the clusters that were processed using the Viterbi algorithm. Clusters with states larger than 8191 (*i.e.,* 12 conflicting methods using formula 2) are processed using the A* algorithm. The processing time for these large clusters is small and as indicated in Figure 7. Moreover, Table 3 shows that the processing time grows linearly.

To measure the memory required to represent the generated states, we examine the space required for one state. The data structure representing one state is illustrated as follows in the class "*State*". Based on the Java data type definition, the space required for keeping one state is approximately 320 bits (40 bytes) (considering "int" and "reference" take 32 bits, and "String" 160 bits for containing 10 "char" types). Therefore, the cluster with 8,191 states requires approximately 310.9Kbytes to keep all the states in the memory. As a result, the state transition approach does not impose a large memory requirement.

```
 public class State {
// state identification
    int stateId;
// id of a conflicting method
    String conflingMethodId;
// id of the class candidate that a conflicting
// method is assigned to
    int assingedClassCandidateId;
// a reference to the previous state
    State preState;
// a reference to the next child state in the
// left branch
    State leftState;
// a reference to the next child state in the
//right branch
    State rightState;
   }
```

4.4 Quality Evaluation of the Migrant Systems

In the last two subsections, we presented results that verified that the presented incremental migration approach is scalable and can migrate medium size legacy procedural systems to object oriented systems. In this section, we verify that the approach can generate object oriented system with the highest maintainability and reusability. To achieve this objective, we conduct the case study as follows:

- Select other possible migrated systems and compare their quality with the migrated system generated from the optimal transformation path.
- To compare the quality of systems, collect a set of object oriented metrics to quantify direct measurable features, for example, coupling between classes and cohesion inside a class. The high cohesion and low coupling further indicate high maintainability and reusability.
- Analyze and compare the metric results from different systems.

To select other possible migrations of the same systems, we use the transformation path presented in our incremental migration approach. All transformation paths are generated and ordered in a sequence based on their attached likelihood scores. Several representative target systems can be generated from different paths. In the experiment, every path is assigned with a number in the range of 0 and 1. Such number shows a relative position of a path in the sequence. For example, the path labeled with 1 refers to the optimal path with the highest likelihood. The path labeled with 0 refers to the worst path with the lowest likelihood. Similarly, the path of 0.5 means the medium path whose likelihood is ranked in the middle of the highest likelihood and the lowest one. The path of 0.75 has the likelihood ranked in the middle of the optimal path and the medium path. Likewise, the path of 0.25 has the likelihood ranked in the middle of the medium path and the worst path. In such a way, we can generate five target systems from the paths of 0, 0.25, 0.5, 0.75 and 1. We can then compare the quality of these generated systems with the quality of the optimal migrated system.

To assess whether the desired goals in the target system have been achieved, a set of widely accepted object oriented software metrics are adopted. In the context of object oriented migration, we aim to achieve high cohesion and low coupling in the target object oriented systems. In this respect, coupling between classes can be measured using metrics, such as, CBO (Coupling Between Objects), DCC (Direct Class Coupling) and IFBC (Information Flow Between Classes)[7]. Moreover, the cohesion inside a class can be measured using metrics, such as, TCC (Tight Class Cohesion), Coh (Cohesion Measurement) and IFIC

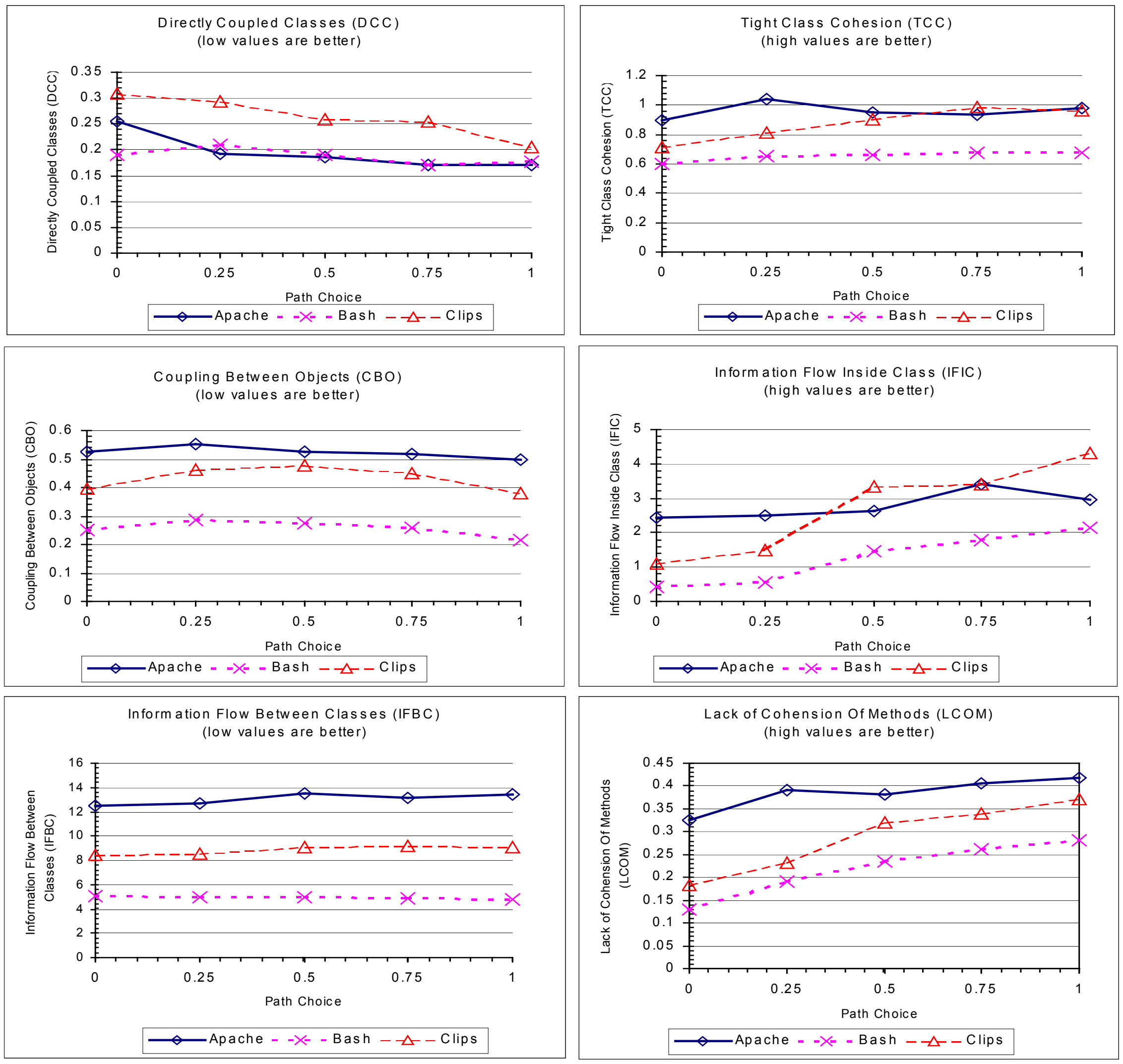

Figure 7. Comparison of Metric Results from Six Metrics

(Information Flow Inside Class) [7]. To reflect the overall value of the entire system in terms of one specific metric, we calculate the average value of the each metric value from each class. These set of metrics are different from the one we used to compute the transformation likelihoods. The overall results from the aforementioned six metrics are illustrated in Figure 7. The X-axis shows the choices of paths. The Y-axis represents metric values corresponding to the systems generated from the five selected paths. One single metric can not reflect the overall quality of a system. For example, in the Figure about the result of IFIC, the Apache system of 1 has a lower IFIC value than the system of 0.75, but the values of TCC and Coh are higher than the system of 0.75. To compare the overall quality between two alternative systems, we count the number of the metric results which are higher in the optimal system than the others. By combining results from all the metrics, we see that the final object oriented system generated from the optimal path has the highest qualities in comparison to the object oriented systems generated from the alternative paths.

5. Related Work

Several methods for identifying an object model from a legacy system have been defined. In [8] an approach to identify object-oriented model from RPG programs is presented. The class identification is centered around persistent data stores, while related chunks of code of the legacy system become candidate methods. In [9], a semi-automatic tool to migrate PL/IX programs into C++ is presented. The abstract syntax tree (AST) is used to analyze the original source code. Consequently, tokens in the AST are transformed into C++. Through a similar approach, an automatic Fortran to C/C++ converter is provided in [10], a Pascal to C++ converter in [11], and methodologies to directly identify C++ classes from procedural code written in C in [12]. Moreover, in [12], an evidence model is presented to help the user select the best-migrated object model. This evidence model includes state change information, return types, and data flow patterns. However, there is no comprehensive framework for ensuring that the migrated system posses certain quality characteristics.

Software quality properties reflect the degree of the conformance to specific non-functional requirement. A process-oriented framework to represent non-functional requirements as soft goals is proposed [4]. The framework consists of five major components: a set of goals for representing nonfunctional requirements that positively or negatively contribute to other goals; a set of link types for relating goals to other languages; a collection of correlation rules for deducing interdependencies between goals; and finally, a labeling procedure which gives a quantitative method to determine any given nonfunctional requirement is being addressed by a set of design decisions. In [13], a quality-driven software re-engineering framework is proposed. The framework aims to associate specific soft goals, such as performance and maintainability with transformations and guide software reengineering process. This work focuses on providing a catalog of transformations to refactor object oriented code to produce better object oriented designs using design patterns. We provide an approach that focuses on the process of software migration of procedural code to object oriented platforms. We aim to refine high level quality goals such as maintainability and reusability into low level source code features (metrics) in the target object oriented systems. Moreover, we devise a quantitative method to assess the likelihood of each transformation to achieve desired quality goals and generate target object oriented systems with the highest quality.

6. Conclusion

This paper presents techniques and methodologies for the incremental migration of procedural legacy systems into object oriented platforms. We define a quality driven migration framework to monitor and evaluate quality attributes at each step of the migration process. Such migration process involves the representation of procedural code using XML, the recovery of the object model from a procedural system, and the incorporation of non-functional requirement into the migration process. A system segmentation technique is provided to decompose a system into a collection of clusters, and consequently divide the migration process into phases. In such a way, a large system can be reengineered gradually in order to reduce the risk and computation costs involved. The approach has been used to migrate a number of open source projects. The obtained results demonstrate the effectiveness, usefulness, and scalability of our framework.

Currently, the proposed framework is applied in migrate legacy systems written in C and Fortran to C++. On-going work focuses on applying the proposed framework in more generic software transformation context, such as refactoring, restructuring and development process.

Acknowledgment

I would like to thank my Ph.D. thesis supervisor, Dr. Kostas Kontogiannis for his valuable contribution to this paper.

References

[1] M.M. Lehman, "Programs, life cycles, and laws of software evolution", Proc. IEEE, v. 68, 1980.

[2] Ying Zou, Kostas Kontoginnis, “A Framework for Migrating Procedural Code to Object-Oriented Platforms”, Proc. of the 8th IEEE Asia-Pacific Software Engineering Conference (APSEC), 2001.

[3] Ying Zou, Kostas Kontogiannis, “Migration to Object Oriented Platforms: A State Transformation Approach”, Proc. of the 19th IEEE International Conference on Software Maintenance 2002.

[4] John Mylopoulos *et. al.*, “Representing and Using Nonfunctional Requirement: A Process-Oriented Approach”, IEEE Transactions on Software Engineering, v. 18, n. 6, June 1992.

[5] Ying Zou, Kostas Kontogiannis, “Quality Driven Transformation Compositions for Object Oriented Migration”, the 9th IEEE Asia Pacific Software Engineering Conference (APSEC), Gold Cost, Queensland, Australia, December 2002 , pp. 346-355.

[6] Ying Zou, Kostas Kontogiannis, “Migrating and Specifying Services for Web Integration”, In Lecture Notes in Computer Science LNCS vol. 1999, Springer-Verlag, 2001.

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

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

[9] Kostas Kontogiannis, et. al., “Code Migration Through Transformations: An Experience Report”, IBM CASCON 1998.

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

[11] “Pascal to C Converter”, http://www.garret.ru/~knizhnik/ptoc/Readme.htm.

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

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