2005

Coping with legacy system migration complexity

Lei Wu

Houari Sahraoui

Petko Valtchev

Follow this and additional works at: http://scholarworks.rit.edu/other

Recommended Citation

Wu, Lei; Sahraoui, Houari; and Valtchev, Petko, "Coping with legacy system migration complexity" (2005). Accessed from http://scholarworks.rit.edu/other/29

This Conference Proceeding is brought to you for free and open access by RIT Scholar Works. It has been accepted for inclusion in Presentations and other scholarship by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.
Coping with Legacy System Migration Complexity

Lei Wu, Houari Sahraoui, Petko Valtchev
CP.6128 Succ. Centre-Ville, Canada, H3C 3J7
{wulei, sahraouh, valtchev}@iro.umontreal.ca

Abstract

During the last three decades, a considerable amount of software has been developed based on obsolete technologies (such as using procedural languages). This type of systems has undergone severe code revisions during a long time period. As a consequence, the high level of entropy combined with imprecise documentation about the design and architecture make the maintenance more difficult, time consuming, and costly. On the other hand, these systems have important economical value; many of them are crucial to their owners [4]. For the high cost of lost former investment and business knowledge embedded in those systems, in many cases, simply abandon legacy systems and re-develop new systems based on new technology is not the choice. Migrating legacy system toward new emerging technology is an appropriate solution. However, migrating legacy system towards new technology is a complex system engineering work. In this paper, we propose a novel approach to reduce the migration complexity. We apply dynamic program analysis, software visualization, knowledge recovery, and divide-and-conquer techniques to cope with the complexity issue in legacy software migration project.

Keywords: legacy software, business logic, architecture recover, migration complexity, decomposition algorithm, fuzzy prioritization

1. Introduction

In recent years, many new computing technologies have emerged, and more are on the way to come in the near future [5]. Meanwhile, during the last three decades, a considerable amount of software has been developed using obsolete technologies, such as procedural languages. For example, Coyle estimates the size of systems written in Cobol to be more than 100 billion LOC [6]. This type of systems has undergone severe code revisions without a real concern of maintaining the documentation up-to-date [7]. As a consequence, the higher level of entropy combined with imprecise documentation about the design and architecture make their maintenance more difficult, time consuming, and costly. On the other hand, these systems have important economical value and many of them are crucial to their owners [4]. For the high cost of lost former investment and business knowledge embedded in those systems, in many cases, simply abandon legacy systems and re-develop new systems based on new technology is not the choice. Migrating legacy system toward new emerging technology is an appropriate solution.

According to particular circumstance, different modern technology may be selected as the migration destination, such as migrating legacy stand-alone system towards web/mobile computing paradigm; or migrating procedural software into object-oriented technology. However, migrating legacy system towards new technology is a complex system engineering work. The difficulties lie in the understanding of target system; the recovery of legacy business knowledge/design architecture; and the methodology of the migration process, etc. All these factors underline the importance of reducing the migration complexity in a re-engineering project.

In this paper, we propose a novel approach to reduce the migration complexity. We apply dynamic program analysis, software visualization, knowledge recovery, and divide-and-conquer techniques to cope with the complexity issue in legacy software migration project. We have also developed analysis tools to support our approach. The paper is structured as follows: in the next section, we present the outline of our approach. In section 3, we mainly illustrate the techniques that we’ve applied to reduce the complexity in legacy system understanding. In section 4, we discuss the methods and algorithms we’ve developed for legacy system decomposition and architecture recovery. In section 5, we demonstrate our fuzzy priority model that applied to the divide-and-conquer method. In section 6, we give the discussion and conclusion of our approach, and point out the future work.

2. Approach Outline
To effectively reduce the complexity of migrating a large legacy software system towards a new technology-based paradigm, we focus on three crucial aspects of our migration methodology.

One is the legacy system understanding. The subject system will be analyzed in various ways to reveal the general sketch of the system architecture. We provide reverse engineering tools to help programmers to accelerate the knowledge recovery process, thus to reduce the complexity of understanding target system, and eventually facilitate legacy system migration.

The other aspect is how to decompose legacy system into small, easy handle, and highly cohesive divisions. To reduce the complexity of a large legacy software migration project, a successful strategy requires an effective mechanism to decompose the system into parts that can be analyzed, modeled and migrated separately. The result of legacy source code decomposition will directly affect the migration efficiency. In fact, decomposition also largely depends on our former research result: the legacy understanding.

The last one is how to organize the migration process in a cost-efficient way, by which can reduce the complexity in dealing with those decomposed parts of target system. This is mainly achieved by prioritizing the migration sequence of individual migration units.

3. Legacy System Understanding

When we face a legacy system migration project, the first question is, “What does the system do?” Programmers are inevitably facing the difficult task to understand legacy software system. Hall’s research work shows that understanding the documentation and logic of programs occupies about 50 to 60 percent of maintenance programmers’ time [1]. In many cases, even the original developers find it is hard to understand their own code after a period of time. As a consequence, migration tasks tend to be expensive, complex and error prone. How to reduce the difficulty and the complexity of legacy system understanding becomes an urgent issue for researchers.

3.1 Business Rule (logic) Extraction

Business logic is operational rules, often coded into software that organizations follow to perform various activities. It includes two types: one is the macro business activity control rules, such as transaction processing, business planning and quality control, etc.; the other one is the micro business regulations, such as data integrity, application domain constraint, business formulas, etc.

One characteristic of legacy system migration is the preservation of business logic. As legacy system has evolved for a long period, many changes are not well- or even not documented. Therefore, the embedded business logics are hard to extract. Consequently this increases the complexity of legacy system migration process. To conquer such problem, one of the most important tasks in our research work is to find out the implementation of business logic inside of legacy code. Because of its intrinsic difficulties, we mainly focus on the recovery of business rule formulations. When migrating a certain MU (Migration Unit), which is the cohesive part of the decomposed result of target system, the embedded business rules should be kept as what they are in the legacy system. Facing this challenge, we’ve developed a technique to extract code fragments that implement business rules based on static data analysis, dynamic behavior analysis and program visualization techniques.

3.2 Static Data Analysis

Business rule can be defined as a function, constraint or transformation of an application’s inputs into its outputs. Formally, a business rule \( R \) can be expressed as \( F(I) \) that transforms a set of input variables \( I \) into a set of output variables \( O=F(I) \). i.e.,

\[
\text{Interest} = (\text{PresentTime} - \text{StartTime}) \times \text{Rates}
\]

Business rule can be defined as a function, constraint or transformation of an application’s inputs into its outputs. Formally, a business rule \( R \) can be expressed as \( F(I) \) that transforms a set of input variables \( I \) into a set of output variables \( O=F(I) \). i.e.,

\[
\text{Interest} = (\text{PresentTime} - \text{StartTime}) \times \text{Rates}
\]

Business rule can be defined as a function, constraint or transformation of an application’s inputs into its outputs. Formally, a business rule \( R \) can be expressed as \( F(I) \) that transforms a set of input variables \( I \) into a set of output variables \( O=F(I) \). i.e.,

\[
\text{Interest} = (\text{PresentTime} - \text{StartTime}) \times \text{Rates}
\]

Business rule can be defined as a function, constraint or transformation of an application’s inputs into its outputs. Formally, a business rule \( R \) can be expressed as \( F(I) \) that transforms a set of input variables \( I \) into a set of output variables \( O=F(I) \). i.e.,

\[
\text{Interest} = (\text{PresentTime} - \text{StartTime}) \times \text{Rates}
\]

Business rule can be defined as a function, constraint or transformation of an application’s inputs into its outputs. Formally, a business rule \( R \) can be expressed as \( F(I) \) that transforms a set of input variables \( I \) into a set of output variables \( O=F(I) \). i.e.,

\[
\text{Interest} = (\text{PresentTime} - \text{StartTime}) \times \text{Rates}
\]

Business rule can be defined as a function, constraint or transformation of an application’s inputs into its outputs. Formally, a business rule \( R \) can be expressed as \( F(I) \) that transforms a set of input variables \( I \) into a set of output variables \( O=F(I) \). i.e.,

\[
\text{Interest} = (\text{PresentTime} - \text{StartTime}) \times \text{Rates}
\]
present them in a more understandable representation forms.

3.3 Behavior Analysis and Program Visualization

We also apply visualization analysis technique to distill business rules by analyzing particular participant code segment through program execution behavior. Most of the business rules have a distinct representation form during the usage of the system [9]. More importantly, the representations of business rules normally have a domain terminology name. All these will either be embodied into system interface or be produced as final computing results. From legacy software’s human-machine interface, programmers and users cooperate together to select which system functionality includes those domain terminology words. By executing those system functionalities and using our dynamic behavior analysis tools: DynamicViewer, programmer can find out which part of the source code has implemented the business rules [10]. With limited size of related code, it will be more efficient to find out code fragments that implement business formulas.

In the functionality analysis of an experiment legacy system, called “Interest” (a legacy financial and stock analysis system), we meet with the following business rules from user interface: stock value, shares, price, net profit, net cost, investment return and internal rate of return. For most of them, we can guess their meanings, but we are still not able to figure out the exact business rules and calculation formulas, such as “internal rate of return”. Our automatic dynamic analysis and visualization toolkit, DynamicViewer, provides two methods to efficiently locate the implementation of those business rules. This helps programmers quickly recover the definition of a specific business rule inside of the huge amount of source code.

The first method is to use the “Hierarchical Module Dispersal & Scale Diagrams” from DynamicViewer to detect the “hot-spots” of the implementation of certain business rules. DynamicViewer visualizes the computation workload of each participation code module in terms of color and size. See figure 2. The row represents the invocation depth; the column indicates each module. When sender module invokes a message to a receiver module, the receiver will lie on one depth below the sender. The rectangle color box represents the quantity of interaction instances at different depths. Its color shows the overall invocation scale of the whole interaction space; the box’s size exhibits the scale compared with all of its own interactions. The line displays the invocation direction from higher depth to lower depth, and its color shows the intensity. Users are lead to the sensitive computing parts shown by the visual result diagrams; explore the computation space by browsing the code segments that related to the module in certain invocation depth; and quickly locate the implementation code of that business rule.

For example, to recover the business rule of “internal rate of return”, programmers may firstly use the “Scenario Recorder” function provided by DynamicViewer to retrieve the user interfaces of this business rule from database, as shown in following figure 1. The left snapshot shows the system user interface before clicking the calculation button; the right picture illustrates the graphical analysis result after clicking calculation button.

![Figure 1: Retrieve dynamic user interface with DynamicViewer](image)

Programmers can use the coarse-grained and fine-grained visualization tools provided by DynamicViewer to detect the hot-spots of the major computing part among the code interactions executed after clicking “Start” button in “Interest” system. And further to recover which modules and routines that heavily contribute the final result shown in right side of figure 1. Meanwhile, a pop up code browser will help programmer to view the related code segments of the related hot-spot source code modules.

As we can see from figure 2 (i), the three bright red modules, namely module 23, 2, 29, located on depth 13 and 14, construct the most calculation work. Therefore, we can quickly locate the implementation code modules of this business rule. The source code collaboration analysis provided by another tool we’ve developed, called Collaboration Investigator [2], will further visualize the source code collaboration patterns that implement this business rule.

The second method is to analysis two business rules, and compares their difference in “Hierarchical
Module Dispersal & Scale” diagrams. The variance normally includes the different implementation code that realizes the different business rules.

(i) “Stock value”

(ii) “Internal rate of return”

Figure2: Business rule comparison detection with DynamicViewer tool set

In Figure 2(i), it shows the hierarchical oscilloscope view of calculating the analysis graph of “stock value” business rule, and (ii) illustrates the view of computing “internal rate of return” business rule. From the disperse shape, color, source code module invocation size and depths, etc, we can immediately figure out these the two visual analysis results are almost identical. This indicates these two rules are similar. Legacy program uses almost the same code modules to perform the similar jobs. But soon, programmers will find out that, in the simulation of “internal rate of return” business rule, see Figure2(ii), there’s a long, vertical and evenly spread green boxes appear at the location of source code module 34 from depth 9 to 17. It reveals a new code module consistently participates in the calculation of “internal rate of return” rule, but does not contribute to the computation of “stock value” business rule. This strongly suggests that, the important implementation code of the “internal rate of return” business rule is located within source code module 34. After analysis, we find that, the “internal rate of return” business rule includes the “stock value” business rule. The calculation of the “internal rate” is in fact exactly located in source code module 34.

4. Legacy System Decomposition and Architecture Recovery

4.1 Module Dependency Analysis

In a legacy system, the user-defined data type and data structure can be viewed as an indicator of the relationship between their host modules and the rest of the system. This is reasonable, since the dependency of user-defined data type reflects the module relationships between each other. Therefore, our source code module dependency analysis has been built based on the user data type dependence relationships. Moreover, the module dependency relationship can further be viewed as an indicator of dividing the whole system source code into organically integrated parts. The borders of constructions are the borders of the module dependency relationships. Consequently, we can decompose the whole system into parts that have high cohesion value according to their module dependency relations. For most legacy procedural languages, the whole system can be divided into many program pieces (such as source files) that are inter-dependent. Normally, the original developers had a certain kind of principle to organize their program artifacts. What’s the rationale behind the organization of these program pieces? Why developers group parts of programs into one unit (such as routines, files), while group other parts into other units? By analyzing the dependency of user defined data type (UDT), we can elicit useful design rational of legacy architecture construction. Here, we look at individual program pieces unit (e.g. files) as a single unit module of source code. The user data type dependency (one UDT uses other UDTs) reflects the reliance relationships of its host modules. Based on this observation, we implement the special tools to automatically construct the source code module dependency diagram, thus greatly reduce the complexity of decomposing target system, and facilitate the recovery of legacy architecture. The following is an elicited module dependency diagram from the experiment legacy system “Interest”. In figure 3(i), the original legacy function call graph shows the whole system source code interaction is like a spaghetti jungle. It’s hard to decompose system according to this type of information. In figure 3(ii),
the source code modules are grouped according to the module dependency analysis.

(i) Legacy system function call graph: a complex spaghetti view of “Interest”.

(ii) Module dependency diagram

Figure 3: Source code decomposition with module dependency analysis

The insufficiency of this method is its accuracy. Designers have their own design/programming styles. We cannot assume all of them create a perfect design, especially for legacy system which has undergone a heavy maintenance work during the long evolving period. Therefore, the question will be like: how much does module dependency diagram faithfully reflect the legacy architecture? Theoretically, module dependency diagram will only faithfully express the design level consideration, but it may not be exactly obeyed in the implementation. We observed that, this method can reflect the design level architectural decomposition of legacy system, but can not assure a faithfully representation at the implementation level, since some modules are only used partially by other modules, the cohesion relationship between two modules can not be assumed to be tight. Meanwhile, it is not applicable to those programming languages that do not have user-defined data types.

4.2 Analysis of Dynamic Information

We apply this technique to analyze legacy source code based on system execution information. It helps programmers analyze the legacy architecture construction, and further facilitates the legacy system decomposition.

Figure 4: Dynamic analysis workflow

A modified legacy source version will be automatically generated; probe code is injected into original source code. By executing the modified version of legacy system, we’ll retrieve the dynamic information of a certain execution test case. Visual effects will be used to map the system behavior with legacy source code. Visualization has been presented at different levels based on different granularity of abstraction.

We have two kinds of view-points towards legacy system. One is the executable software system; one is the legacy source code. Both can be abstracted into a layered form.

We divide program artifacts into several layers. Each layer has a certain degree of program abstraction. The executable system’s logic layer division can be viewed as the whole system, subsystem, subsystem functions, detail functionalities, etc, see Figure 5(i).
While the physical program source code layers can be viewed as program piece units [8] (source code files/modules), functional component parts, detail routines, variables, data structures etc, see Figure 5 (ii). What we have is the legacy program artifacts, including program files, data files (or database files), documents, etc. Then, we try to answer the question of “which part of the program artifacts realize which functionality that provided by legacy system?” Mapping the system functionalities with the realizing code fragment will clearly divide the whole source code into organic parts, thus reveal the construction of legacy software architecture. Based on this idea, we have developed tools to record the active system functionality behavior and the corresponding executed code fragment that participates in such activities, and show their consequent relations in an animated manner. The most important feature of this technique is the mapping ability for different abstraction layers. Generally, a fine-grained code artifact’s behavior mapping will make programmer get lost inside of the huge information swamp, and limits this technique as a debug level tool. By mapping particular system functionality with different abstract layer of program artifacts, a more efficient high level view of system structure will be revealed. According to OMG, an architecture can be recursively decomposed into parts that interact through interfaces; relationships that connect parts; and constraints for assembling parts [3]. Therefore, applying this technique, we will be able to map the system functionality with different program abstract parts, recover the relationships within their interfaces and calculate the dynamic program metrics and visualize the calibers of the system structure.

When a certain type of system functionality is performed, each abstraction map will show the behavior trace. The highest source code module interaction graph will show the dynamic communication trace of that atomic functionality execution. Therefore, the modules on the trace record will be formed into one group to generate a new set. While the lowest level entities (routine, global variable, etc.) also can be grouped according to the behavior trace. When different functionality is performed, the behavior trace will change, thus the elements of group sets will also be changed. Dynamic behavior analysis will automatically check the changes of each trace set, distill the constant elements that always participate in the activity when executing a certain kind of system functions, thus to find out the kernel part that realize the system behavior. Therefore, we can decompose the whole system based on system functionality, and according to this decomposition, we are able to decompose legacy source code correspondently based on applying our dynamic visualization analysis technique. Our dynamic analysis tools produce two kinds of different useful results:

1. **Module interaction diagram.**
   Which modules are always working together to perform a certain kind of system functionality, thus they are belong to one subsystem or component. This information will help programmer finally depict the architecture of the legacy system

2. **Routine interaction diagram.**
   This will show inside of module, which routines are always interact together to perform a certain kind of atomic system functionality. Thus they can be grouped within a class entity. This information will help expert recover candidate classes.

Different system functional layer and source code layer will be used to simulate the information routine by applying color, brightness, progress route highlight etc. These will simulate the dynamic reflection between system unit functionality and source code abstraction layers, thus to facilitate the legacy system decomposition task. This technique also helps us find out much useful source code integrity information. For a certain kind of system atomic functionality, by executing different test cases, we’ll get different sets of participated abstract layer’s fragments. The tool also calculates which part of each different abstract layer has persistently participated; which are conditionally involved; which bear the heaviest computing burden while which parts work as information deliver, etc.

### 4.3 Dynamic Connection Algorithm

The source code module is represented as nodes in the interaction graph. The dynamic decomposition strategy is as follows: for a certain system function, we automate the retrieval of module invocation diagram dynamically. The code modules appeared in the graph
illustrates the contribution to the behavior of this system function. See figure 6. The horizontal line represents participant modules, named as Mi (i=1,2…); the vertical line represents the invocation level.

**Figure 6: Dynamic Interaction Graph**

The linkage between modules shows a certain degree of cohesion. We use following algorithm to calculate the cohesion metrics among any two modules that appeared in the interaction graph.

```plaintext
For (i=1 to Module_Number)
    For (j =1 to Module_Number)
        IF (i!=j)   THEN   Cohesion_0 (Mi, Mj)= 1;
//{Arbitrary assign each pair of modules in the graph has 1 degree of cohesion values.}
For (difference =1 to Total_Level -1)
    For (Each Pair(Mi, Mj) in graph )
        IF   Distance (Mi,Mj)=difference  && Link (Mi,Mj)
            THEN
                Cohesion_difference(Mi,Mj)=Total_Level-
difference;
    For (i=1 to Module_Number)
    For (j=1 to Module_Number)
        For (k=0 to Total_Level – 1)
            If (i<>j &&  Cohesion_k(Mi,Mj)>0)
                Cohesion(Mi, Mj)+=Cohesion_k(Mi,Mj);
        Group modules according to Cohesion(i,j) values>Threshold;
    Return (decomposed module sets);
```

**Table 1: Dynamic Connection Algorithm**

The modules have a certain degree of cohesion, suppose the maximum invocation number is M, now we assume it is 10. Arbitrary we assign each pair of them has 1 out of 10 (M) degree cohesion values if they have any kind of connection. Further, for each pair of code modules, if they have direct link in the graph (function call), then they have as high as 9 (10-1=9) degree cohesion value for this pair. Accordingly, if they have two level linkage, such as M1->M2->M5, then {M1,M5} has a (10-2=8) degree of cohesion value. We monitor system function execution and record all the invocation graph, then calculate the sum of cohesion value degree for each code module pair, and continue such merge with second level pair, until reach a certain low level threshold of cohesion value degree. By this way, we’ll obtain a divided code module group set. Each set has a number of different code modules, among which have reasonably high degrees of cohesion values. Performing this algorithm will generate a decomposition of legacy system.

### 4.4 Static Decomposition Algorithm

**Terminology:** the following terms will be used throughout this section. First, we call each source file in a legacy software system as a module. Modules will be clustered into groups that can be migrated at one step of the process, i.e., processed by a single team within reasonable time limits, called Migration Units (MU). To fix MU dependencies between modules, e.g., service delivery, a dependency graph is constituted where vertices are modules and the service requests (invocations) are directed edges. An intermediate step is the factoring of that graph into strongly connected components (SCC) (i.e., maximal groups of modules that depend on each other, either directly or by transitivity), and the construction of the respective directed acyclic graph (DAG) of all SCC. The MU then emerges as subsets of the levels in the DAG. Here, a level m is defined as the set of graph nodes for which the longest path from (one of) the universal source node(s) (no in-going edges) in the DAG is of length m.

**Finding MUs in Legacy Software:** a key task is the split of the entire system into tractable parts which are both (i) logically connected and (ii) admit a single-step migration. As a first approximation that fits (i), one may identify MUs with modules. However, most modules are of smaller size so that a module-based decomposition may lead to a large number of individual migration tasks and, thus, to a high synchronization cost. Therefore, it is more convenient to define MUs as sets of modules, whereby a preferred size for MUs becomes an important parameter of the decomposition that insures (ii). However, condition (ii) imposes some restrictions on the way modules are grouped into MUs. In fact, whenever two modules A and B are linked in the graph, say A depends on B, or AÆB, it is preferable to migrate B before migrating A in order to avoid some reworking of the already migrated code imposed by the decision A before B. This means that if the duration of migration tasks is a concern, one should consider (ideally) only MUs made
up of modules with no dependencies. Such groups may be identified by looking at the module dependency graph, as defined by previous section, where candidate MUs are independent subsets of graph nodes. Unfortunately, in a large number of cases, the information provided by the graph may not be sufficient. Indeed, the strict respect of inter-module dependencies may be hindered by the presence of graph circuits, i.e., configurations in which sets of modules depend on each other, either directly or by transitivity. In this case, neither of the concerned modules may be migrated before the others without some post-reworking. The solution seems to reside in the joint migration of all the modules. However, the resulting task is of a much higher complexity than the migration of an equal number of unrelated modules. Therefore, the number of such tasks should be kept to a strict minimum. To formalize the underlying problem, we apply a graph-theoretical framework based on the concept of SCC. The approach is summarized as follows

Build module dependency graph $G$;  
Apply discovery algorithm to detect the set $S$ of all SCC and to organize $S$ in the DAG $D$;  
i = 0;  
While $S$ is not empty  
Level[i] = all nodes in $D$ with no in-going edges (no service to other nodes);  
Erase the nodes of Level[i] from $S$ and $D$;  
i++;  
For (j = 0 to i-1)  
IF (the size of each node in Level[i] < threshold size)  
Then  
Group nodes into a minimal set of MUs so that each MU size < the threshold;  
ELSE  
FOR (Each node whose size > threshold size)  
Apply slice-merge strategies to split the node into smaller nodes (modules);  
Group the resulting nodes into a minimal set of MUs of size < the threshold;  
Return the set of discovered MUs;

<table>
<thead>
<tr>
<th>Table 2: MUs Construction: Static Decomposition Algorithm</th>
</tr>
</thead>
</table>

The SCC of the dependency graph is considered instead of single nodes, i.e., modules, and the respective factor-graph, the DAG of the SCC, is examined for MUs. The sets of independent nodes in the DAG are drawn from its level-wise split: the highest level, e.g. 0, is made up of all nodes (if any) that do not provide services to the outer world (universal clients), while the nodes of level $n+1$ are those which provide services only to nodes of levels 0 to $n$.

5. Prioritizing Migration Process

Since legacy systems are normally large in size, progressive migration will largely reduce the reengineering complexity, and require relatively smaller resources at a single point of time. After successfully decompose legacy system, we’ll face the problem of which MU(Migration Unit) should be migrated first. For a certain MU decomposition diagram, we firstly analyze its characteristic, then compare with available reengineering resources, finally adopt a suitable priority strategy, apply it on all of the MUs, and select the best candidate to migrate first.
5.1 Module Unit Migration Complexity Metric

Priority Criteria defines what elements we should consider to rate the migration priority of a group of migration units (MUs). Module Complexity Metrics are applied here to determine the complexity degree of each MU. We defined following metrics to represent the measurements of module complexity model.

Isolation degree: relationship with other MUs is automatically calculated in form of reference number (weights). In another word, the less relation with the rest of the system, the higher isolation degree it will have;

Module importance: by the help of dynamic behavior analysis tools, those MUs that perform higher important tasks of the system will be ranked a higher number of importance;

Understandability: a subjective analysis and other source code metrics combined together to indicate the understandability degree of a particular MU, the comparative difficult order will reflect the priority sequence;

Feasibility of applying automatic modeling techniques: a number of randomly selected sample code segments will be used to test the feasibility of using automatic techniques to elicit new models (based on target new technology) from legacy code. The easier the MU has, the higher feasibility degree it will have, thus reflects its priority order.

Environment independency: if a given MU relies little on any specific application environment (such as OS, hardware, language, etc.), it will have a higher degree of environment independency. For example, if a MU has a large percentage of code implementing a language related GUI part, when the target system use another language which incompatible with the original interface API, then, most of the legacy code which use the special API will be discard. Therefore, the higher environment independency degree a MU has, the higher migration priority it should be assigned.

5.2 Priority Fuzzy Synthesis

The above criteria may not be coincident all together, we only use their combination to produce a synthetic one. We apply fuzzy expert system to indicate the final sequence priority order. It is an expert system that uses fuzzy logic instead of Boolean logic. In another word, a fuzzy expert system is a collection of membership functions and rules that are used to reason about data. Unlike conventional expert systems, which are mainly symbolic reasoning engines, fuzzy expert systems are oriented toward numerical processing. The rules in a fuzzy expert system are usually similar to the following:

\[
\text{if } x \text{ is low and } y \text{ is high then } z = \text{medium}
\]

where \(x\) and \(y\) are input variables, \(z\) is an output variable, low/high/medium are membership functions defined on \(x\), \(y\) and \(z\) respectively. The part of the rule between the "if" and "then" is the rule's premise which describes to what degree the rule is applicable. The part of the rule following the "then" is the rule's conclusion that assigns a membership function to each output variables. For each criteria defined above, we can define a member function. For example, if the highest module importance is defined as 10 and lowest is 0, we may define the member function as illustrated in Figure 8(i).

In the inference process, the truth value for the premise of each rule is computed, and applied to the conclusion. We'll get a synthetic priority result by applying all the rules. Here we simply illustrate how to apply fuzzy logic to combine two criteria, see figure 8(ii): Model Complexity and Isolation degree to produce the final migration priority degree. In our real work, we apply all the criteria to indicate the final MU priority.

![Fuzzy member function](image1)

![Defuzification to generate final result](image2)

**Figure 8:** Fuzzy synthesis of migration priority by two criteria

Migration Priority Iteration: after migrating the MU that has the highest migration priority, we can iterate the procedure to produce the next candidate for
migration in the next stage. Since many situation has changed, we need re-evaluate the priority order which is produced in former stage, and produce new priority order for the rest of MUs, and select the MU which has the highest migration priority, and iterate the migration process. With the help of prioritizing migration units, we divided the whole process into small and easy handle sequence of migration activities, thus reduce the complexity of one-bang approach.

6. Conclusion and Future Work

Migrating legacy system towards new technology is a complex system engineering work. The difficulties lie in the understanding of target system; the recovery of legacy business knowledge/design architecture; and the methodology of the migration process, etc. All these factors underline the importance of reducing the migration complexity in a re-engineering project. In this paper, we have presented a novel approach to reduce the legacy system migration complexity. We apply dynamic program analysis, software visualization, knowledge recovery, and divide-and-conquer techniques to cope with the complexity issue in legacy software migration project. We have also illustrated the analysis tools set that we’ve developed, to support our approach. The initial experience shows the result is promising. Our approach has demonstrated the feasibility and utility of using various techniques to cope with legacy migration complexity issue. In the future, we are continuing this work by improving the analysis functions of our tools, and conducting more experimental studies.

Acknowledgements: This research is funded by “Nature and Technology Research Funds” of Quebec, Canada.

References


