Organizing imperative programs for execution-based retrieval for reuse.

June Xuejun. Qiang
University of Windsor

Follow this and additional works at: https://scholar.uwindsor.ca/etd

Recommended Citation
https://scholar.uwindsor.ca/etd/2966

This online database contains the full-text of PhD dissertations and Masters' theses of University of Windsor students from 1954 forward. These documents are made available for personal study and research purposes only, in accordance with the Canadian Copyright Act and the Creative Commons license—CC BY-NC-ND (Attribution, Non-Commercial, No Derivative Works). Under this license, works must always be attributed to the copyright holder (original author), cannot be used for any commercial purposes, and may not be altered. Any other use would require the permission of the copyright holder. Students may inquire about withdrawing their dissertation and/or thesis from this database. For additional inquiries, please contact the repository administrator via email (scholarship@uwindsor.ca) or by telephone at 519-253-3000ext. 3208.
INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films
the text directly from the original or copy submitted. Thus, some thesis and
dissertation copies are in typewriter face, while others may be from any type of
computer printer.

The quality of this reproduction is dependent upon the quality of the
copy submitted. Broken or indistinct print, colored or poor quality illustrations
and photographs, print bleedthrough, substandard margins, and improper
alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript
and there are missing pages, these will be noted. Also, if unauthorized
copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by
sectioning the original, beginning at the upper left-hand corner and continuing
from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced
xerographically in this copy. Higher quality 6" x 9" black and white
photographic prints are available for any photographs or illustrations appearing
in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®
ORGANIZING IMPERATIVE PROGRAMS FOR
EXECUTION-BASED RETRIEVAL
FOR REUSE

by
June Xuejun Qiang

A Thesis
Submitted to the College of Graduate Studies and Research
through Computer Science
in Partial Fulfillment of the Requirements for
the Degree of the Master of Science at the
University of Windsor

Windsor, Ontario, Canada
1999
© 1999 June Xuejun Qiang
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author’s permission.
Abstract

As libraries of reusable software components continue to grow, the issue of retrieving components from the library has become more and more important. In order to make software reuse possible, the library should collect enough components to support developers, but when the number of component increase, finding and choosing an appropriate component becomes more and more difficult. A few retrieval techniques have been used to address the problem of finding relevant components, but how effectively the library is structured and how the appropriate components are found quickly have received considerably less attention.

In the thesis, we propose a method for organizing imperative program components in a library for supporting execution-based retrieval. A prototype system, called OERS, is developed using OOD and OOP.

This system could be used to construct and maintain an organized reuse library through the initiation, deletion, adaptation, and browse operations. A retrieval subsystem allows user to retrieve, test and view components.

OERS system provides more efficient retrieval method in terms of the retrieval time. The benefit is directly proportional to the total number of components in the library. Based on our experiments, the average retrieval time is 7 times faster than general execution-based retrieval[32].
To my parents

my husband Jie Zhu and my son Calvin

for their love and support
Acknowledgements

First of all, I want to thank all the people who helped me during my study and thesis work. These are our professions, stuff and graduate students.

I am indebted to Dr. Young Park who provided help and advice on both the courses and the thesis. Without his dedication and guidance, I could not finish this work.

I am deeply appreciative of Dr. Hu for her being my external reader.

Special thanks also to Dr. Chen for her suggestions and comments, and to Dr. Jaekel for her time.

Finally, a big thank-you to my parents, who helped me to take care of my son; to my husband Jie Zhu, for his understanding and encouragement; to my son Calvin for his patient during the long hours I spent on my thesis.

I am so sorry that I did not have much time with my father during his final days, I am sure he would be happy to see I have finished this thesis.
# TABLE OF CONTENTS

Abstract iii

Acknowledgements v

List of Figures ix

List of Tables xii

Chapter One  Introduction 1

1.1 Concepts of Software Reuse 1

1.2 Common Software Reuse Techniques 5

1.2.1 Generative Approach 5

1.2.2 Compositional Approach 6

1.3 Component Retrieval 8

1.4 Imperative Programming Language 11

1.5 Overview of the Thesis 12

1.5.1 Motivation 12

1.5.2 The Objective of the Thesis 13

1.5.3 Organization of the Thesis 15

Chapter Two  An Approach for Organizing Library 16

For Execution-based Retrieval 16

2.1 General Concepts of Organizing the Library 17

2.2 Insertion of Components 21

2.3 Deletion of Components 23

2.4 Retrieval the Desired Component 26
Chapter Three  A Prototype System (OERS) Design

And Implementation  28

3.1 System Design  28

3.2 Algorithms for Organizing and Retrieval  36

3.2.1 Initial the Sub-library  36

3.2.2 Insertion of Components into Sub-library  37

3.2.3 Adaptation the Sub-library  40

3.2.4 Deletion of Components from Sub-library  41

3.2.5 Browse the Library  42

3.2.6 Retrieval of Component from the Library  43

3.2.7 Test the Candidate Function  45

3.3 Implementation  46

3.3.1 System Implementation  46

3.3.2 Integrating Native Code with Java Code  47

3.4 Interface Implementation  55

3.4.1 OERS Manager Interfaces  55

3.4.2 OERS Retrieval System Interfaces  61

Chapter Four  Examples, Evaluation and Conclusion  64

4.1 Examples of Using OERS  64

4.2 Evaluation of the Approach and Implementation  74

4.3 Conclusion  76
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>The Model of Execution-based Retrieval</td>
<td>9</td>
</tr>
<tr>
<td>1.2</td>
<td>Prototype System Model</td>
<td>14</td>
</tr>
<tr>
<td>2.1</td>
<td>The Structure of the Library</td>
<td>18</td>
</tr>
<tr>
<td>2.2</td>
<td>The Structure of the Sub-library</td>
<td>19</td>
</tr>
<tr>
<td>2.3</td>
<td>Another Structure of the Sub-library</td>
<td>20</td>
</tr>
<tr>
<td>2.4</td>
<td>Insertion of Components into Sub-library</td>
<td>22</td>
</tr>
<tr>
<td>2.5</td>
<td>Deletion of Components from Sub-library</td>
<td>24</td>
</tr>
<tr>
<td>2.6</td>
<td>Deletion of Last Component from Sub-library</td>
<td>25</td>
</tr>
<tr>
<td>2.7</td>
<td>Searching Components in the Organized Sub-library</td>
<td>27</td>
</tr>
<tr>
<td>3.1</td>
<td>OERS Use Case Diagram</td>
<td>28</td>
</tr>
<tr>
<td>3.2</td>
<td>Library Manager Use Case Diagram</td>
<td>29</td>
</tr>
<tr>
<td>3.3</td>
<td>OERS User Use Case Diagram</td>
<td>30</td>
</tr>
<tr>
<td>3.4</td>
<td>High-Level Library Manager Class Diagram</td>
<td>30</td>
</tr>
<tr>
<td>3.5</td>
<td>High-Level Retrieval System Class Diagram</td>
<td>31</td>
</tr>
<tr>
<td>3.6</td>
<td>Detailed Library Class Diagram</td>
<td>32</td>
</tr>
<tr>
<td>3.7</td>
<td>libraryInf Class Diagram</td>
<td>33</td>
</tr>
<tr>
<td>3.8</td>
<td>Detailed libUtil Class Diagram</td>
<td>34</td>
</tr>
<tr>
<td>3.9</td>
<td>Detailed Execute Class Diagram</td>
<td>34</td>
</tr>
<tr>
<td>3.10</td>
<td>Initial Algorithm</td>
<td>37</td>
</tr>
<tr>
<td>3.11</td>
<td>Detail Flowchart</td>
<td>39</td>
</tr>
<tr>
<td>3.12</td>
<td>Adaptation Algorithm</td>
<td>40</td>
</tr>
<tr>
<td>3.13</td>
<td>Deletion Algorithm</td>
<td>41</td>
</tr>
</tbody>
</table>
Figure 4.9  Retrieval in subLib12  71
Figure 4.10 Some Components in subLib21  72
Figure 4.11 Retrieval in subLib21  73
Figure 4.12 Some Components in subLib21 After Adapting  73
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 3.1</td>
<td>Primitive Native Types</td>
<td>50</td>
</tr>
<tr>
<td>Table 3.2</td>
<td>Reference Native Types</td>
<td>50</td>
</tr>
<tr>
<td>Table 3.3</td>
<td>Primitive Array Types</td>
<td>51</td>
</tr>
<tr>
<td>Table 3.4</td>
<td>Type Signature Encoding</td>
<td>53</td>
</tr>
<tr>
<td>Table 4.1</td>
<td>Initial the Library subLib11</td>
<td>65</td>
</tr>
<tr>
<td>Table 4.2</td>
<td>Library sublib11 after adding 9 functions</td>
<td>66</td>
</tr>
<tr>
<td>Table 4.3</td>
<td>Compare Retrieval Time</td>
<td>74</td>
</tr>
</tbody>
</table>
1.1 Concepts of Software Reuse

Software reuse is the process of implementing or updating software systems using existing software components.

The concept of reuse is simple and nature. When people meet a new problem, they usually try to solve the problem using the solutions of similar problems, if the solution does not fit the problem completely, they will try to modified them. Proven solutions, used over and over to solve the same type of problem, become accepted, generalized, and standardized. Formal generic mathematical models are good examples of successful reuse because they can be applied to solve specific problems across several engineering fields or domains.

There are two concepts underlying reuse. First, avoiding reinventing those that have already been invented; second, using already tested and proven components as building blocks to construct new systems or modify existing system.

However, components refer not only to source-code fragments, but also to all the products generated during software development, such as requirements, designs, models, algorithms, tests, documents, and any information the developer needs to create software [1,2].
Reuse may occur within a software system; across similar software systems; or in widely different software systems (e.g., user interfaces for thousands of different application packages).

Software development with and/or for reuse has offered following advantages:

- **Increase Software Productivity**
  
  Because software reuse uses created, tested and documented components to build new system, developers need to do less work, and thus, productivity is improved.

- **Improved reliability**
  
  Because used components are tested and proved in real systems. They are more reliable than those new components which tested in simulated test environments. Confidence in reused components is therefore higher. Ideally, components should be used without change to have confidence in their reliability.

- **Rapid system delivery**
  
  Time to market is the most critical factor for some software systems, especially among the competitive companies. Who first delivers the product, who may rule the market. Reusing components reduces the time required for system development and delivery. It also makes Prototyping possible.

- **Potentially lower costs**
  
  The cheapest component is the one that you do not have to write. NASA quotes a 4:1 return on investment in reuse. Validation costs in embedded systems are great than 50% of costs: cost of inspection and review, Cost of developing test data, cost of running test data. Major validation cost savings is possible. Major development and maintenance cost savings are possible through reuse but there are additional costs. Component retrieval
costs, component change cost, architecture modification costs, component generation costs.

- Risk reduction

Reusing a component rather than re-implementing it reduces management risk. It is less likely that the development will meet unexpected technical problems, and it is easier to estimate the costs, effort and time from starting the work to having an operation implementation.

Many companies around the world are reporting successful reuse programs.

In AT&T in 1988, a domain-specific, large-scale software-bus system was developed for on-line transaction processing and network management. Their reuse program has reduce development costs by about 12 percent and time-to-market from 18-24 months to 6-9 months. In IBM, they have several reuse support centers, a large parts library, and a multisite Corporate Reuse Council. One of two departments at HP[4] is the Manufacturing Productivity section of HP’s Software Technology Division which started reuse program in 1983 by using generated code and other work products such as application and architecture utilities and files. The other is San Diego Technical Graphics which develops, enhances, and maintains firmware for plotters and printers. The program began in 1987. The Boeing Co., British-Telecommunications, PLC, Eastman Kodak Company, Ford Aerospace Corporation, Motorola, The National Aeronautics and Space Administration (NASA), and many others also introduced the reuse programs in their companies.
Foley\textsuperscript{[2]}, director of systems and technology planning at one of North America's largest grocery companies, expects a 30% productivity gain by using component-based development.

Raytheon\textsuperscript{[2]} saw a 50% increase in productivity in its Missile System Division. The Navy experienced a 26% reduction in required labor hours to develop and maintain its Restructured Naval Tactical Data System.

It appears the product development costs, factoring in the cost of producing, supporting, and integrating reusable software components, can decrease by a sustainable 10 to 20 percent; defect rates in delivered products can be dropped drastically to 10 percent of their former levels; and long-term maintenance costs can be dropped to 20 to 50 percent of their former values when several products share the same, high-quality components\textsuperscript{[14]}. 
1.2 Common Software Reuse Techniques

Usually, software reuse techniques can be divided into two categories: generative approach and compositional approach.

1.2.1 Generative Approach

Generative approach is reuse at the specification level by means of application generators. Levy\textsuperscript{[6]} proposed a coarse three-step methodology for developing software system by application generators.

a. Identify the requirements for the generator.

b. Building the generator.

c. Using the generator.

Application generators operate like programming language compilers, that inputs a set of specifications and outputs the code of an application within an implementation language, but it is different from compilers: its input specifications are typically very high-level, special-purpose abstractions from a very narrow application domain.

The best-well known examples of application generators are Lex and Yacc for Unix system. Lex deals with lexical analysis, the input of the Lex is a regular expression language that describes the legal tokens (identifiers, keywords, constants, etc.) in the language to be compiled. Yacc is used for parsing. A context-free grammar, which defines the syntax of the language to be compiled, is used as input.
In application generators, algorithms and data structures are automatically selected so the software developer can concentrate on what the system should do rather than how it is done. It means nonprofessional programmers to create expected software could use the application generators. And an application generator takes the system specification as input and produces executable code as output. Software developers do not have to test that the generated code satisfies the system specification, because the implementers have verified that the code is always consistent with the input specifications.

Generally, compared to conventional software development, application generator reduces some steps in software development lifecycles for the software developer by moving much of the development effort into the hidden and automated parts of the generator.

But because the specification language used depends very heavily on the application domain, it is difficult to design a development methodology for application generators that is appropriate for all application generators. Building generators is another challenge due to the difficulties in recognizing appropriate cases, defining the requirements as well as validating the outputs. According to the literature, there are only a few application generators that are available, many of which have narrow domain coverage. So, the generative approach has not been adopted by practicing programmers.

1.2.2 Compositional Approach

Compositional reuse is using the existing components as building blocks to create systems. Although as mentioned before those components can be any products during
the software developments, but in practice, compositional reuse has focused mainly on source code reuse.

Usually there are 3 phrases in this approach. First, a reuse library is organized, and the components are stored. The library often supports multiple versions of the components. Second, some support tools or methods are used for retrieving and identifying the desired components. If the exactly desired components do not exist, there is one more phase: component adaptation: it is to retrieve and modify most closed components to fit the requirements. After we have got all the desired components, the final state is component integration.

This approach was first introduced by McIlroy[6] in 1968. With the development of software engineering techniques, many modern programming languages and paradigms provide a lot of features to support for building general and thus highly reusable software components. The object oriented programming is the best example.

The most advantage of this approach is the potential capability to cover an entire domain. But there are also a lot of difficulties. For example, most software library like the IMSL math library[6], uses keywords or texts to describe components but unfortunately most programming language do not come with abstraction mechanisms for describing component behavior and for doing classification and retrieval.

In my thesis work composition approach is adopted.
1.3 Component Retrieval

From discussion above, we can see that retrieving reusable components is the first task in the compositional approach. To select a component for reuse, you must know what it does. This is addressed by how to describe the components.

In addition to retrieving the correct components, a component library must provide software developers with techniques to locate components efficiently. Without such technique, a developer must search through a large, monolithic collection or less structured reusable components. With the growth of the library size, effective and efficient retrieval of candidate components becomes more and more important.

Most existing retrieval approaches are based on the syntactic or semantic descriptions.

- Syntactic retrieval includes enumerated classification, faceted and free-text indexing.[23]

Enumerated classification is a well-known retrieval method consisting of a hierarchy of inclusion relationships between categories. Faceted classification classified the components by choosing values for a set of attributes called facets. It is more flexible than hierarchical schemes. Free-text indexing methods use the text from a document for indexing. Users specify a query using key words to find matching documents.

The strength of these methods is that people can use the knowledge contained in the structures to find the information they want. The weakness is that no support is possible if the information is not structured properly. Also, users may have the difficulties to build keywords or terms due to the ambiguity of nature languages and difference of individual. [30]
- Semantic-based retrieval can be divided into three groups: type-based retrieval, execution-based retrieval and formal specification-based retrieval.

Formal Specification uses well-defined syntax to describe the component. The syntax and semantics of specification are used to direct the search for a component. The accuracy is high, but it is very hard to implement and requires the developers to be familiar with the specification languages [1,5,21].

Type-based retrieval uses component type as key to search the component. The response may have a lot of components. So the accuracy is low, but it can give a big cut to the library components. Developers can further use other method to get more accurate result from the approximate result. Similar to Type-based retrieval, Semantic-properties-based retrieval uses a number of semantic properties such as the length property on list arguments, demand property of arguments as well as others to retrieval components. It is also approximation retrieval [13,14].

Execution-based retrieval uses the behavior of the executable components as the key to retrieval, because every component should have its own unique behavior even sometimes the difference is small, so this method is expected to achieve higher accuracy [3,14,17,32].

The model of execution-based retrieval is illustrated in figure 1.1.

![Diagram](image)

Figure 1.1 The Model of Execution-based Retrieval
A pair of sample input and corresponding desired output of the potential candidate components is sent to the search engine. The search engine executes all components in the library using the sample input, then retrieval the components whose output is same as the desired output.

But this approach also has shortcomings: it is expensive, for performing one search, all the components will be executed regardless they are relative or not. And it uses random sampling to generate input needlessly reduces the precision.

From above, we can see the effectiveness problem becomes more important in this approach.
1.4 Imperative Programming Language

Most of the popular languages have been designed to suit the popular computer architecture called the Von Neumann architecture since 1940's. These languages are called imperative languages like basic, Fortran and c.

Imperative programming languages are based on the concept of commands, which is the statement used to update variables held in storage. Because the languages have close relationship with machine architecture, they can be implemented very efficiently and quite natural to modern real-world problem.

But such programming languages have side effects. Like in C, arguments or parameters can be changed during the execution of function/procedure.

For examples:

a. void convToLow (char *s) {

    while (*s != '\0') {
        if (*s >= 'A' && *s <= 'Z')
            *s += 32;
        ++s;
    }
}

b. void convToUp (char *s) {

    while (*s != '\0') {
        if (*s >= 'a' && *s <= 'z')
            *s -= 32;
        ++s;
    }
}

These two procedures with same type, i.e., char* -> void, but the values pointed by the pointer have changed and may be different after they are executed, although both of them may have the same argument and return null.
1.5 Overview of the Thesis

1.5.1 Motivation

As libraries of reusable software components continue to grow, the issue of retrieving components from the library has become more and more important. In order for reuse possible, the library should collect enough components to support developers, but when the number of component increase, finding and choosing an appropriate component becomes more and more difficult. Retrieval techniques as mentioned before have been used to address the problem of finding relevant components, but how effective the library is structured and how the appropriate components are found quickly have received considerably less attention.

Dr. Park and his previous students P. Bai, K. Khalil and D. Ramjisihgh have designed and implemented a prototype system called Wiser (Windsor Internet Software-basE of Reuse). In the system, Function components in Miranda can be retrieved by Execution.

K. Tam, another student of Dr. Park, extended the approaches to retrieval imperative programs, such as C programs. But in this approach, the library was not structured and the drawback of Execution-based retrieval mentioned before still remain.

In 1996, Dr. Park presented a method of organizing existing reusable functional programming code components into a library. But it is not implemented and not for imperative languages.
1.5.2 The Objective of the Thesis

In my thesis, a method that extends the method presented by Dr. Park\textsuperscript{[12]} is proposed to organize the imperative programs in the Library for execution-based retrieval.

- The basic idea is that use space trade-off time. Since in execution-based retrieval, given a query, all components in the Library are executed during the search for obtaining the actual output. So, the search is time consuming and not practice. To shorten down the search time, we can save the query and actual outputs in the library, So that next time, we just search the stored query and results rather than re-execute the components.

- Because the components are imperative programs, arguments or parameters can be changed during execution of the function/procedure. The output values for the same input is not enough to describe the behaviors of the components. to capture the behaviors of such components, the final states of the arguments are treated as part of behaviors of the components.

- Combining the classification and execution-based methods. We first divide the whole library into sub-library according the signature of the function/procedures. Each sub-library only contains the same type function/procedures. This will cut down the irrelative components. Then we use execution-based retrieval to find desired components.
• The method provides a way that allows the user dynamically test the candidate components by any proper input data without executing other components in the same sub-library.

• A prototype system is implemented using client/server model. The whole system can be expressed like below:

![Diagram of Prototype System Model](image)

Figure 1.2 Prototype System Model

Client sends a query to server, server starts to search by search engine, then sends the results back to client side.

Use the server/client model has following advantages:

It splits the whole process into parts, each part become small and manageable and handles its own problem. If a bug in the server side is triggered, the server side may be crash, but this will not usually bring the client side down.

A client communicates with a server by sending it message. The client does not know whether the message is handled locally in its own machine, or whether it was sent across a network to a server on a remote machine. It means this model is adaptability to be used in distributed system.

• Java language is selected as the implement language because it is object-oriented language. After carefully building basic, all-purpose classes, you can easily create special-purpose system using the built classes. It is built with network
communications in mind. So it is possible to be further extended to distributed system. It is also highly portable. Java system can be ported to almost any computing environment such as SUN Solaris, Windows, and Macintosh.

- The system provides two tools for finding the desired components. One is for library managers to organize all components in the library by providing initial, insert, delete, browse and adapt skills; another is for users who can retrieval the components by retrieval, test and browse skills. Such separation makes the system more simple, easy to use and maintainable.

1.5.3 Organization of the Thesis

This thesis is organized into four chapters.

Chapter 1 gives a brief introduction of relative concepts in the software reuse. Such as common approaches in software reuse, common techniques in component retrieval, benefits by using reuse etc. Finally overview of the thesis is addressed.

Chapter 2 discusses the detail of the proposed method: how to structure the library and how to retrieve desired components based on such organized library. Some examples are provided for explanation.

Chapter 3 gives the details of the design and implementation of the prototype system. UML is used as a tool to describe the functionality of the system and the static structure of the system. Java Native Interface(JNI) is used to perform execution of C functions in Java environment.

Chapter 4 gives a few samples on using OERS, and compares the proposed methodology with the generalized execution-based retrieval\textsuperscript{[32]}. It also concludes problems left and the future works in this approach.
Chapter Two

An Approach for Organizing Library for Execution-based Retrieval

To incorporate reusable components into system, we must find components first. If this process fails, then reuse cannot happen. In chapter one, several retrieval techniques have been discussed, execution-based retrieval is expected to achieve higher precision than other approaches because every component should have its own unique behavior.

But in the general execution-based retrieval, all components are actually executed during retrieval process. The retrieval time depends on the execution time and would be too long to be incorporated in practice\(^{[12]}\). So the effectiveness and efficiency become the big issues in this approach.

Even with innovative and intelligent retrieval tools, a reuse library’s effectiveness is directly proportional to the quality of the indexing and structures for retrieval.\(^{[23]}\)

In this chapter, we describe the approach to organizing library as well as the retrieval based on such structure. The basic idea is memorization. We store the actual behavior of the function and the input values at first time by executing the function for the input data. When the function is re-applied to the input data later, we could reuse the result stored before instead of re-executing the function for the same input data.
2.1 General Concepts of Organizing the Library

We introduce some concepts which may be used in the further discussion.

Type of the library: all the components (functions in C at here) with same type of arguments and return type are put in the same library. The type of arguments and return type together is called the type of the library, also called the type of the component.

For instance, type of function: void convToUp(char *s) is char * -> void;

    type of function: int max(int x, int y, int z) is int, int, int -> int

Post argument statues (values): after executing the function, all the argument values are called post argument values. For example: if the input value for function convToUp is a pointer which points to string "low", after execution the string it should become "LOW". this value is the post argument values.

Behavior of the function: we treat the responses of a function for certain inputs as the behavior of the function. So the post argument values and return value of the function to an input sample data are called the behavior of the function for the input data, simply behavior of the function.

For example:

    { "LOW" ; void} is the behavior of the function convToUp for input "low".

    { "LOW" ; void} is also the behavior of the function convToUp but for input "LOW".

    { "low" ; void} is the behavior of the function convToLow for input "low".
{ "low" ; void } is also the behavior of the function convToLower but for input "LOW".

From above examples we could see, the function such as convToUpper may have the same behaviors \{ "LOW" ; void \} for the different input data like "low" and "LOW", but for one input data, the behavior is unique and certain. Different functions usually have different behaviors for same input data like "LOW".

A match means that the user desired behavior is same as the actual behavior of the function for the same input data. For instance: if the user hopes to find a function which could convert string from low case into upper case, and after getting the actual behavior of the function convToUpper \{ " LOW" , void \} for input "low", which is just what the user wanted behavior, then we say this is a match. We assume that two floating point numbers are "equal" if their absolute difference is less than a small value.

Before memorization of the behaviors of the components on sample input into the library, we divide the library into sub-library, each sub-library has one type.

![Figure 2.1 The Structure of the Library](image)

-18-
The type information and corresponding sub-library’s names are stored in a text file. The correct sub-library in which candidate component resides could be found by using the type information as the key to checkup in the text file.

Each sub-library has a tree structure storing the components with their behaviors for some input data. The root node is the first sample input. The other nodes are pairs of behavior responding to their parent nodes’ previous sample inputs and current sample input. The leaves are pairs of behavior responding their parent nodes’ sample inputs and the name of the component.

Consider the following components are in one sub-library:

```c
void iniStr(char *); /* initialize the pointer to blank string */
void printStr(char *); /* print the string the pointer pointed */
void convToUp(char *); /* convert the string to uppercase */
void convToLow(char *); /* convert the string to lowercase */
void firstUp(char *); /* convert the first char in the string to uppercase */
```

the organized sub-library may look like:

![Diagram of the Sub-library Structure]

Figure 2.2 The Structure of the Sub-library
If we select sample input different from the first one, for example like "HE", the structure of the sub-library may become:

![Diagram of another structure of the sub-library]

Figure 2.3 Another Structure of the Sub-library

When we retrieval the component, we just search a match between the desired behavior and actual behavior stored in the library by traveling the tree, which means that actual execution of the components is done once only when it is needed.

Suppose we get the desired components directly, then the average steps for getting all the components in above sub-library are 1.4 and 2 for first structure and second structure respectively. That means how to select the inputs is very important to achieve good structure.

To explaining clearly, we give illustrations to describe how the library is built through component insertion, deletion, and how to retrieval components in such structured library in the later sections.
2.2 Insertion of Components

When first component goes to the Library, user is asked to provide a sample input as the root node.

When adding new component into existing sub-library, the system uses the root node as the sample input for the new component. If the actual behavior is different from any existing behavior, then it adds the behavior and new component’s name as the leaf, otherwise, it selects the leaf which has same behavior as current one, change the leaf to node with the behavior and all the names of components.

If the node has more than two children with same behaviors, we try to find another sample input which will cause at least one component in the node with different behavior, split the node into one node and one (or more) leaf.

If this node has the same behavior for many sample inputs (e.g. exceed a fix number like 10), then we treat all components in the node as the “same components”.
Figure 2.4 Insertion of Components into Sub-library
2.3 Deletion of Components

The process of deleting component from the structured sub-library process is described in figure 2.5.
When deleting a leaf, if the left leaf has two nodes above and the directly node has no other children, the direct node and the leaf are needed to be merged as one leaf, otherwise just simply remove the leaf.
When only one component is left in the sub-library, the deletion process not only removes the leaf, but also needs to empty the root node.

Figure 2.6  Deletion of Last Component from Sub-library
2.4 Retrieving the Desired Component

Retrieving the desired component is the process of searching a match in the structured sub-library without doing actual executions.

First, user needs to provide the behavior of the desired component should have for the input data at the root.

Then the system searches the direct children to check if there is a match. If no match found, "no matched components found!" message is return. Otherwise, the system checks the matched child is node or leaf. If it is leaf, then it returns the function name as candidate component, otherwise the system asks user to give another desired behavior corresponding the new input data and repeat above steps, until the candidate is found or return "no match components found" message.

For example:

If user is looking for a function which assigns a char pointer pointing to an empty string, the desired behavior for the input value at root node ( "HE" ) is { " " ; void}. When the system searches the match in the sub-library, it occurs at first leaf (match found along the left arc arrow path).

If user hopes the candidate function can print out the string pointed by a char pointer, the behavior for input "HE" should be { " HE" , void" }, but there are two functions with the same behavior for such input, so one more input
is introduced, the new desired behavior for the new input "He" is \{ "He", void \}. Along the right arc arrow path, the match is found, So printStr is retrieved!

Figure 2.7 Searching Components in the Organized Sub-library
The proposed **Organized Execution-based Retrieval System (OERS)** is designed by UML, implemented by Java. Reuse by OERS is on the source code level. All the components in the library are C functions.

### 3.1 System Design

The UML (**Unified Modeling Language**) is a language that unifies the industry’s best engineering practices for modeling system. Based on the object-oriented paradigm. It is used here for specifying, visualizing, constructing and documenting systems.

![Figure 3.1 OERS Use Case Diagram](image-url)
As shown in Figure 3.1, OERS use case diagram describes the functionality of the system and users of the system. OERS has two main services: one for the library manager to organize the library, another one for OERS user to perform the retrieve on the organized library.

![Library Manager Use Case Diagram](image)

Figure 3.2 Library Manager Use Case Diagram

Figure 3.2 elaborates the library manager use case by detailing the functionality a library manager actor expects of the system.

Library manager may initial, insert, and delete the components. He/she also can browse all the components in the library; adapt or improve the structure of the library. Library inf & util use case is called or used in all the skills use cases and organized library.

Figure 3.3 elaborates the OERS user use case by detailing the functionality a user actor expects of the system. He/she could browse the functions stored in the library,
retrieve the function wanted. Further test the retrieved function can be achieved by test skill. These three skills use library Inf & util use case, that also associated with organized library.

Figure 3.3  OERS User Use Case Diagram

Figure 3.4 describes the static structure of library manager system on how it is structured rather than how it behaves.

Figure 3.4 High-Level Library Manager Class Diagram
Library system is concerned with initializing library, inserting a component into library, deleting component from library, adapting the structure of library, browsing all components in the library and displaying help information. The filled diamond indicates a composition relationship. For example: the filled diamond attached to the adapt class of the association between the adapt class and the LibraryInf class indicates that the adapt contain GroupInf class.

Figure 3.5 describes the high-level retrieval system class diagram. In this system, client/server model is used to pass message. OERS user provides retrieval information to retrievalClient, then the client passes the query to retrievalServer. The server uses this information to find the candidate components from the organized library and returns the result to retrievalClient. Once the desired component is found, the user can test its

---

**Figure 3.5 High-Level Retrieval System Class Diagram**
behavior by providing more input test data to testClient. When testServer gets the new test data, it will invoke exec method in execute class to obtain the actual behavior and return result to testClient.

Figure 3.6 is the detailed library class diagram. Because all the components are C functions, they need to be declared as native methods in Java class before they could be invoked. One sub-library class contains all the functions with same type. For example: subLib11 contains all the functions with type: int -> long; subLib21 contains all the functions with type: double double -> double.

subLib is an abstract class: it uses attribute postArgs to store the post argument values, setPostArgs() is an abstract method, it will be implemented by its subclass. All subLib* (like subLib11,subLib12, ...) inheritances subLib and must implement method setPostArgs.

![Detailed Library Class Diagram](image)

Figure 3.6 Detailed Library Class Diagram
Figure 3.7 describes the libraryInf class diagram. As mentioned before, all the mapping information between sub-library's name and the type is stored in a text file. libraryInf class provides the tools for reading, writing, and deleting such information in the test file. The fileName in the methods is the name of the text file containing mapping information.

![libraryInf Class Diagram]

Figure 3.7 libraryInf Class Diagram

Each sub-library holds the structured components, and a tree represents this structure. Each tree is stored in a text file (for example subLib11, subLib21, ...). Class libUtil provides the tools to manipulate this text file. Please see Figure 3.8 for detail.
Class execute is a very important class. It is the core for executing C function in Java program.
It dynamically invokes C function. At the compiler time, we don’t know which function needs to be executed, so an attribute funName is used to hold the function name. Before invoking the function, all the information, such as the type of the function and input data, are also needed known by the environment. This can be done by passing values to argTypes, returnType, inputArgs attributes and calling setInputVals method.
3.2 Algorithms for Organization and Retrieval

For simplicity and conveniences, we use the tree to describe each sub-library. The tree is stored in a text file, and each node/leaf has a fixed length. Every node or leaf in the tree represents following information:

< Parent index; children indexes; node or leave; behavior; sample input values or function names>

shorting written as <PI; CI; N; BI; V/ FNs>.

The root of the tree only has children indexes and first sample input values.

For other parts, if it is node, the last item is another new input values; otherwise, it is collection of function names with the same behavior for the sample input values given at the parent node.

3.2.1 Initialize the Sub-library:

When the first function is added into the library, we need to create a sub-library, then store the input data and function name as well as the behavior as first two rows. This phase is called initialize the sub-library.

For example: when we first insert a function: long IntSqrt( int) into the library, we need these steps.

Given: type of the library T; sample input values V; function name FN; sub-library L

Start:
get the name of the sub-library by using the type information. Suppose it is subLib1.

Check if subLib1 exists?
if exists, check if it is empty or not?

If not empty, that means subLib1 has been initialized, user should check again.
If empty, go to initial.

if not exists, create a new text file called subLib1.

Initial: invoke FN on V and get its behavior.
set first record root and its child: first leaf in subLib1.
3.2.2 Insertion of Components Into Sub-library

Insertion is the very important skill in the system. It decides the structure of the library. As mentioned before, different first test data will cause different structure, so we should try to select the test data which could express the most properties of the functions.

Please note that actual execution is only needed for the new component. The basic idea is described here:
Given: type of the function T; function name F

start: get the sub-library name L by using the type T.

Let root-node in L as current_node.

step1: get current_input from current_node.

step2: get current_behavior by executing F on current_input.

check current behavior with the behavior of each child of the current_node.

if
current_behavior is different from any child’s behavior

then create a new child (leaf) under current_node

else update current_node with the child_node which has the same behavior.

if

current_node is actually a leaf

then add F into the function names contained in the current_node.

else repeat from step2.

Figure 3.11 give the more detail flowchart.
Figure 3.11 Insertion Algorithm
3.2.3 Adaptation the Sub-library

Different functions may have same behavior for some input values. When inserting the functions into the sub-library, if this situation happens, all these functions are grouped together and treated as they are equivalent. But they actually are not the same. To solve this problem, we need more test data to distinguish them. This is why adaptation is needed.

Given: type T of sub-library L; function index FI in which functions are needed to be distinguished. new input values V.

Figures 3.12 Adaptation Algorithm
3.2.4 Deletion of Component from Sub-library

When the function is not needed anymore, or the sub-library needs to be re-built, or the function needs to be modified, we should remove the function from the library.

Given: function type T; Name (function name) wanted to be deleted; and the function index FI

![Flowchart](image)

Figure 3.13 Deletion Algorithm
3.2.5 Browse the Library

Browse is the skill that the user can use to view all the component in the library. User first provides the type of the sub-library, then all the functions’ names and indexes in this sub-library will display. User can view different types of the sub-library by providing the type information after pressing reset button. Once the user select one function, the body of the function will be displayed.

Given: type $T$ of the Sub-library $L$.

![Flowchart of Browse Algorithm](image)

Figure 3.14 Browse Algorithm
3.2.6 Retrieval of Component from the Library

Retrieval of component from the library is the key skill of the whole system. All other skills are finally service for efficient retrieval. User could use this skill to retrieval the desired component by searching the organized library without actually executing it.

Client/server model is adopted to perform this skill. User can perform retrieval even if the library is not on the local computer. First user provides the function’s type information at client side, then this information is passed to server. When server gets the type information, it checks the library and tries to find the corresponding sub-library. After opening the sub-library, server gets the first input test data stored at root node and sends it back to client side. User must provide the desired behavior of the candidate functions for this test data. Server invokes the retrieval skill to compare the desired behavior with the stored behaviors. When a match is found, all the functions in that leaf will be retrieved, the result will be sent back to client side. Client side then displays the result to user.

During the process, user may need to provide a few desired behavior of the candidate functions. Some functions may have same behaviors for some input test data. When this occurs, multiple input test data will be used to distinguish them. On the other hand, a function can be implemented by many ways, so some functions do have same behavior for any input test data. At this circumstance all these functions will be retrieved because they are stored in the same leaf. Figure 3.15 shows the retrieval algorithm at server side.
Given: type T of function, user desired behavior UDB

Figure 3.15 Retrieval Algorithm
3.2.7 Test the Candidate Function

This skill is also implemented by client/server model. User may want to test the candidate function using more test data after retrieving the function from the library. This task needs the server actually execute the function using new test data which are different from the one stored in the library. A process flowchart is given below:

Given: type of the function (T), name of the function (Name), and test data (data)

![Flowchart](image)

Figure 3.16 The Process of Test Skill
3.3 Implementation

3.3.1 System Implementation

The whole system is implemented on window 95 operation system and uses Java Native Interface API (release with JDK2) to support execution of C functions in Java environments. The C functions are built by using Microsoft’s edit and compiler, specifically visual C++ 5.0 and cl.exe. In order to run cl.exe from the command line, the vcvars32.bat batch file shipped with visual C++ is used to set up the appropriate environment. The overall system is illustrated in Figure 3.17.

![Diagram of System Implementation](image)

Figure 3.17 Overall the System
3.3.2 Integrating Native Code With Java Code

For integrating native method (C function at here) in the Java program, you must follow 6 steps.

1. Write Java code which includes C functions.
2. Run Java class file through `javac` to produce class file.
3. Run Java class file through `javah` to produce function prototypes.
4. Write C source code using functions prototype generated by 3.
5. Compile and link C source code to produce object files and then produce a dynamically loadable library.
6. Run `java` on class file produced in 1.

![Diagram](image)

Figure 3.18 Integrating C Code with Java Code
write the java code with C function.

When writing a method implementation in a language other than Java like C, the keyword native is used to tell the Java compiler that the method is a native language function.

After C compile compiles the C function into a shared library, method system.loadLibrary is used to load the shared library into the Java class that requires it.

The following Java code segment defines a class named subLib11. This class declares a few C functions as native methods and loads the shared library.

class subLib11 {
    ......
    public native long IntSqrt( int n);
    public native long IntSquare(int n);
    public native long IntCube(int n);
    ......

    static {
        System.loadLibrary("subLib11");
    }
    ......
}

compile the Java source file.

Once the java source file is created, javac is used to generate the class files. For example, to generate Java bytecode for the java class subLib11, the following command is used:

> javac subLib11.java

generate function prototype

using javah to produce a C header file that defines an ANSI C function prototype
for each native method declared in the input class file, the generated header file is available for inclusion by the C source file containing the implementation of the native methods.

For above class, use the command:

> javah -jni subLib11

the output file will be named subLib11.h. Below is the automatic generated file(segment).

/* DO NOT EDIT THIS FILE - it is machine generated */

#include <jni.h>

/* Header for class subLib11 */

#ifndef Included_subLib11
#define Included_subLib11
#ifdef __cplusplus
extern "C" {
#endif

/*
 * Class: subLib11
 * Method: IntSqrt
 * Signature: (I)J
 */
JNIEXPORT jlong JNICALL Java_subLib11_IntSqrt (JNIEnv *, jobject, jint);

/*
 * Class: subLib11
 * Method: IntSquare
 * Signature: (I)J
 */
JNIEXPORT jlong JNICALL Java_subLib11_IntSquare (JNIEnv *, jobject, jint);

......
#ifdef __cplusplus
}
#endif
#endif

In this header file, the signature for implementation for the native method IntSquare is:

JNIEXPORT jlong JNICALL Java_subLib11_IntSquare (JNIEnv *, jobject, jint)

The name of the C function in C source is Java_subLib11_IntSquare, generally the name consists of four parts: prefix Java, the package name, the class name, and the
native method name, between two parts underscore "_" is used as separator. In above
code, no package name appears because the code is in the default package, with no name.

Please also notice, JNI requires every native method at least having two
parameters in the C source code. First one is JNIEnv interface pointer, the native code
accesses arguments and objects passed to it from Java or send to Java from native code
through this pointer. Second one is jobject, which references the current object itself.
The following arguments are those argument(s) in native method. As above, the argument
int in the native method IntSquare corresponds to the argument jint in the C source code.

The JNI defines the following native reference types and their corresponding Java
reference types. For C compilations, all the native types are typedef from jobject that
defined in <jni.h>.

Table 3.1 Primitive Native Types

<table>
<thead>
<tr>
<th>Native Type</th>
<th>Java Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>jbyte</td>
<td>Byte</td>
</tr>
<tr>
<td>jshort</td>
<td>short</td>
</tr>
<tr>
<td>jint</td>
<td>int</td>
</tr>
<tr>
<td>jlong</td>
<td>long</td>
</tr>
<tr>
<td>jfloat</td>
<td>float</td>
</tr>
<tr>
<td>jdouble</td>
<td>double</td>
</tr>
<tr>
<td>jchar</td>
<td>char</td>
</tr>
<tr>
<td>jboolean</td>
<td>boolean</td>
</tr>
<tr>
<td>void</td>
<td>void</td>
</tr>
</tbody>
</table>

Table 3.2 Reference Native Types

<table>
<thead>
<tr>
<th>Native Type</th>
<th>Java Referent</th>
</tr>
</thead>
<tbody>
<tr>
<td>jobject</td>
<td>object</td>
</tr>
<tr>
<td>jclass</td>
<td>Class</td>
</tr>
<tr>
<td>jstring</td>
<td>String</td>
</tr>
<tr>
<td>Jarray</td>
<td>array</td>
</tr>
<tr>
<td>jthrowable</td>
<td>Throwable</td>
</tr>
</tbody>
</table>
Table 3.3 Primitive Array Types

<table>
<thead>
<tr>
<th>Native Array Type</th>
<th>Java Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>jobjectArray</td>
<td>Object[]</td>
</tr>
<tr>
<td>jbyteArray</td>
<td>byte[]</td>
</tr>
<tr>
<td>jshortArray</td>
<td>short[]</td>
</tr>
<tr>
<td>jintArray</td>
<td>int[]</td>
</tr>
<tr>
<td>jlongArray</td>
<td>long[]</td>
</tr>
<tr>
<td>jfloatArray</td>
<td>float[]</td>
</tr>
<tr>
<td>jdoubleArray</td>
<td>double[]</td>
</tr>
<tr>
<td>jcharArray</td>
<td>char[]</td>
</tr>
<tr>
<td>jbooleanArray</td>
<td>boolean[]</td>
</tr>
</tbody>
</table>

- Write the C source file.

When writing the C code, the type of each actual argument passed to a native method must be one of the native types described above. The type of each argument passed to all JNI functions must also be one of these native types. All returned values from native code must be one of the native types. Any native operations not involved in returning a value or making a JNI function call may use regular ANSI C types.

Every native method receives a JNIEnv pointer as its first argument. This pointer points another pointer which points to a table of function pointers. Each entry in this table points to a JNI function. Specific JNI function could be called in native code to access Java objects.

![JNI Interface Pointer](image)

Figure 3.19 JNI Environment Structure
In our native code, we concern how to access instance java member variables using the JNI interface functions. Because when C function executes on the test data, we not only want to get the output, but also want to know the post argument values, So before exiting from the C function, we need to set the post argument values back to the instance variables in the java class. For example, it is `private int postArgSt` in subLib11 class.

Here is an example to do this:

1. #include <stdio.h>
2. #include <math.h>
3. #include <ctype.h>
4. #include "subLib11.h"
5. JNIEXPORT jlong JNICALL Java_subLib11_IntSquare(JNIEnv * jne, jobject job, jint n) {
6.   jclass thisClass;
7.   jfieldID fid;
8.   int temp = (int)n * (int)n;
9.   thisClass = (*jne) -> GetObjectClass(jne, job);
10.  fid = (*jne)->GetFieldID(jne, thisClass, "postArgSt","I");
11.  (*jne)->SetIntField(jne, job, fid, (int)n);
12.  return ((jlong) temp);

In line 9, we first use the JNI function `GetObjectClass` given a Java object reference. It returns a reference to the class of which that object is an instance. Then we specify the member variable signatures according to the encoding scheme listed at table 3.4. The general form of a member variable signature is:

"member variable type"
The signature for the member variable \textit{int postArgSt} in class subLib11 is "I".

The signature for a Java object, such as a String, begins with letter L, followed by the full-qualified class for the object, and terminated by a semicolon (;).

"Ljava/lang/String;"

Table 3.4 Type Signature Encoding

<table>
<thead>
<tr>
<th>Type Description</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>B</td>
</tr>
<tr>
<td>char</td>
<td>C</td>
</tr>
<tr>
<td>double</td>
<td>D</td>
</tr>
<tr>
<td>float</td>
<td>F</td>
</tr>
<tr>
<td>int</td>
<td>I</td>
</tr>
<tr>
<td>long</td>
<td>J</td>
</tr>
<tr>
<td>short</td>
<td>S</td>
</tr>
<tr>
<td>void</td>
<td>V</td>
</tr>
<tr>
<td>boolean</td>
<td>Z</td>
</tr>
<tr>
<td>fully-qualified class</td>
<td>L&lt;fully-qualified-class;</td>
</tr>
</tbody>
</table>

Once obtained the member variable’s class, name and type signature, we use another JNI function \textit{GetFieldID} to get the identifier for the member variable in line10.

Finally in line 11, another JNI function \textit{SetIntField} is used to send the post argument value (int n in native code) back to the member variable "PostArgSt" in class subLib11. The JNI interface pointer \textit{JNIEnv}, Java object reference \textit{job} as well as the member variable identify \textit{fid} are the arguments of the function.
• Compile the C source file and link the object file to produce a dynamically loaded library by enter the command:

> cl - I c:\jdk\include - I c:\jdk\include\win32 - LD file.c -Fe***.dll

in this command, you need to specify the include path that corresponds to the setup on your own machine. file.c is your native code file. *** in the last item must replaced with the library name written in the statement System.loadLibrary("***") in the Java file.

• After finished above steps, we could run this Java class in which C functions are its native methods.

> java class_file
3.4 Interface Implementation

There are two main interfaces, one for OERS manager system, another one is for OERS retrieval system.

3.4.1 OERS Manager Interfaces

The main manager system interface controls all other interfaces used in the system: initialize the library, insert component, adapt the library, delete component, browse the components in the library as well as provide the help information. These operations are aligned at the left side in the main interface. When user presses one button, another window will pop up, this specific process could be done in this window.

![OERS Library Manager](image)

**OERS Manager**

In this manager, you have following six choices:

- **Initial** --- Use to initial the library. When you insert the first function, you need use this button.
- **Insert** --- When insert new function, please press this button.
- **Adapt** --- Sometimes, there are more than two functions have same return Value and the arguments have same statues after execution for some input sample values, so they are treated as one group. If you want to further divide these functions by using more new input values, use this button.
- **Delete** --- When you want to remove the function from the Library.
- **Browse** --- You use this button to select the library, all the functions in the library will be listed, and if you want to see the body of the function, just click it and the body will display at the next window.

Figure 3.20 OERS Library Manager Interface
In interface initialize, user needs to provide function name, function type information and the test data (at most the number of arguments is five), then press Apply button. If initial success, success information will show in the result area. Otherwise the error information will appear. user could use Reset button to clear the information entered before, and try again.

![Initial the Library Interface](image)

*Figure 3.21* Initial the Library Interface
Insertion interface is similar to initial interface, but user only needs to provide function type information and function name. The stored test data in the library will be used as the input data for this function. Same as initial interface, the maximum number of arguments is five in this interface.

Figure 3.22  Insertion of Component Interface
In deletion interface, user needs to enter function type information, function name, and the function index in the library. Index is a position information of the function in the library. User could find it by browser skill. Using the index information could make the process very efficient.

![Delete function from the library](image)

Please Select type of the library and give the function name and index you want to delete

| Args Types: |  | Return Types: | void |
| function Name: |  | Function Index: |  |

RESULT

![Result](image)

Figure 3.23 Deletion of Component from Library

In adapt interface, use could improve the structure of the library by providing more test data.
For example, if the test data for function \textit{IntSquare} and \textit{IntSqrt} is int 1, then both of them have the same behavior: post argument value is int 1, return long 1, so they are stored in the same leaf and will be retrieved as same candidate components. This is very poor organization. Using adaptation, user provides another test data: int 4, then \textit{IntSquare} will return int 16, \textit{IntSqrt} will return 2. Their behavior is different, and they will be stored as different leaves. That means the structure of the library has been improved.

User could use \textit{Reset} button to clear the information entered before, then use \textit{Try} button to find better result as many times as he/she likes. Once the result is satisfied user can use the \textit{Accept} button at the bottom to keep this result and stored them in the library.

![Adapt the Library](image)

- \textit{Try} - \textit{Reset}

*Please Select type of the library and give the test Input and you can try as much as you want, once the result is satisfied please press ACCEPT button!*

Arg Types: \hspace{2cm} Return Types: \hspace{2cm} Function Index:

New Test Input: \hspace{2cm}

Results:

<table>
<thead>
<tr>
<th>function name</th>
<th>return value</th>
<th>post args value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Accept

\textit{Figure 3.24 Adaptation of the Library}
The Browse interface, let user view all the signatures of functions in the library. Once specific function is selected, the body of the function will show in the *Function Body* area.

![Figure 3.25 Browse the Library](image-url)
3.4.2 OERS Retrieval System Interfaces

The main retrieval system interface controls three other interface: retrieval, test and browse. Brief help information in the interface can help user understand what he/she can do in this system.

**OERS Retrieval System**

In this system, you have following three choices:

Retrieve , Test and Browse.

After you press one button, another window will pop up. Please then fill up the required data and make the action. Good Luck!

Retrieve --- Use to retrieval desired function. you need first to give the signature of the function. then the system will query what return value you want for the given input value.

Test --- After you retrieved the candidate function, you could use this button to test this function by providing new test data, you could test as many times as you like.

Browse --- You use this button to select the library, all the functions in the library will be listed, and if you want to see the body of the function, just click it and the body will display at the next window.

Figure 3.26  OERS Retrieval System Interface
Retrieve component from the library interface is a very important one. All the interactive parts between user and the system, such as input data and action buttons, are on the left side of the window. The main place is used to display help information during the process.

User first enters the type information about the candidate function. The input data will show up in *InputValues* after *Start* button is pressed. Second, the user provides the desired behavior of candidate function for this input data, then press *Continue*. Two cases can occur at this point, one is that the desired function is retrieved, the other one is that a new test data show up again. User needs to repeat above steps until the candidate function is retrieved or "no match" message appears.

![Retrieve Function from Library Interface](image)

Figure 3.27 Retrieve Function from Library Interface
Test retrieved function by new input data interface provides user a flexible tool, by which user could use his/her own test data to examine the retrieved function other than the test data stored in the library. The snapshot is given at figure 3.27.

Browse interface in this system is the same as the one in the manager system, Please see figure 3.24.

![Test Retrieved Function by New Input Data](image)

*Please Select type of the library and give the test input and you can try as much as you want. after test please press Quit button !*

Args Types:  
New Test Input:  
Return Types:  
Function Name:  

Results:

| return value | post args value |

Figure 3.28  Test Components Interface
Chapter Four

Examples, Evaluation and Conclusion

4.1 Examples of Using OERS

- Build and maintain the library

Suppose we have a group of functions with type: int -> long:

long IntSqrt(int);
long IntCube(int);
long InrSquare(int);
long poundToKg(int);
long kgToPound(int);
long IntTenPower(int);
long IntFabs(int);
long Factorial(int);
long CelsToFahr(int);
long FahrToCels(int);

When first IntSqrt is added to library, we use initial skill. If input data is int 16, after executing the function on this input value, the content of the library (called subLib11) lists in table 4.1. At node/leaf column, 0 means this record is a leaf; 1 means this record is node, new input data is required to divide functions in this node.

<table>
<thead>
<tr>
<th>index</th>
<th>parent index</th>
<th>childrens index</th>
<th>node/leaf</th>
<th>return value</th>
<th>postArgs values</th>
<th>input values or function names</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>---</td>
<td>1</td>
<td>1</td>
<td></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>IntSqrt</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1 Initial the Library sublib11

Then we use insertion skill to add all of the rest of functions to subLib11. Table
4.2 lists the content of sublib11 at this moment.

<table>
<thead>
<tr>
<th>index</th>
<th>parent index</th>
<th>childrens' index</th>
<th>node/leaf</th>
<th>return value</th>
<th>postArgs values</th>
<th>Input values or Function names</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>---</td>
<td>1 2 3 4 5 6 7 8</td>
<td>1</td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td>2</td>
<td>64</td>
<td>4</td>
<td>IntSqrt</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td></td>
<td>0</td>
<td>16</td>
<td>4</td>
<td>IntCube</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td></td>
<td>0</td>
<td>10000</td>
<td>4</td>
<td>IntSquare</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td></td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>IntTenPower</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td></td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>IntFabs</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td></td>
<td>0</td>
<td>9</td>
<td>4</td>
<td>KgToPound</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td></td>
<td>0</td>
<td>39</td>
<td>4</td>
<td>CelsToFahr</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td></td>
<td>0</td>
<td>-29</td>
<td>4</td>
<td>FahrToCels</td>
</tr>
</tbody>
</table>

Table 4.2 Library subLib11 After Adding 9 Functions

![Browser Library]

Please Select type of the Library:

Args Types: int
Return Types: long

<table>
<thead>
<tr>
<th>Index</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>IntSqrt</td>
</tr>
<tr>
<td>2</td>
<td>IntCube</td>
</tr>
<tr>
<td>3</td>
<td>IntSquare</td>
</tr>
<tr>
<td>4</td>
<td>IntTenPower</td>
</tr>
<tr>
<td>5</td>
<td>IntFabs</td>
</tr>
<tr>
<td>6</td>
<td>KgToPound</td>
</tr>
<tr>
<td>7</td>
<td>CelsToFahr</td>
</tr>
<tr>
<td>8</td>
<td>FahrToCels</td>
</tr>
</tbody>
</table>

**Function Body:**

Figure 4.1 The Library After Adding 9 Functions

Figure 4.1 is a snapshot of the library. Figure 4.2 is the snapshot of the library after CelsToFahr is deleted.
Figure 4.2 The Library After Deleting One Function

From table 4.2 we can see functions IntSqrt and poundToKg have the same behavior for input value int 4, but they do have different functionality. So adaptation skill is used to distinguish them. Figure 4.3 is a snapshot of Adaptation of the library. Figure 4.4 is the library after adaptation. We could see with one more test data, these two functions have been divided as different groups.
Please Select type of the Library and give the test input and you can try as much as you want, after test please press ACCEPT button!

Args Types: int
New Test Input: 9
Return Types: long
Function Name: 1

Results:

<table>
<thead>
<tr>
<th>function name</th>
<th>return value</th>
<th>post args value</th>
</tr>
</thead>
<tbody>
<tr>
<td>IntSqrt</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>poundToKg</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

Accept

Figure 4.3 Adaptation of the Library
Figure 4.4 The Library After Adaptation

- Retrieve and test the desired components.

Figure 4.5 is a snapshot of retrieval function \texttt{IntSqrt} at the moment when the library has not been adapted. Because \texttt{IntSqrt} and \texttt{poundToKg} are in same leaf, so both of them are retrieved.
Figure 4.5 Retrieval Example 1

Figure 4.6 is a snapshot of retrieval function $IntSqrt$ at the moment when the library has been adapted. So only $IntSqrt$ is retrieved. Figure 4.7 is a snapshot of using test skill to exam $IntSqrt$ with more test data.

Figure 4.6 Retrieval Example 2
Please select type of the library and give the test input and you can try as much as you want, after test please press Quit button!

Args Types: int
New Test Input: 10000
Return Types: long
Function Name: IntSqrt

Results:

<table>
<thead>
<tr>
<th>return value</th>
<th>post args value</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>10000</td>
</tr>
</tbody>
</table>

Figure 4.7 Test IntSqrt with New Test Data

Please select type of the library:

Args Types: char_pointer
Return Types: void

Index  | Function
---|---
1  | convToIntUp
2  | firstUp
3  | firstWordUp
4  | getFirstWord
5  | getTail
6  | reverse
7  | convToLow
8  | trimStr

Function Body:

void convToIntUp (char **)

Figure 4.8 Library subLib12
SubLib12 is a library with type: char pointer → void. All components are procedures. Procedure returns nothing but changes the statues of its arguments. Figure 4.8 is the snapshot of the library by using browse skill. Figure 4.9 is the snapshot of retrieval performed by users.

![Retrieve function from Library](image)

Figure 4.9 Retrieval in subLib12

User first supplies the type of desired candidate, then the desired behavior. In first retrieval, the desired behavior of the candidate is {null; “He is good” }, \textit{firstUp} is retrieved. Second time, {null; “HE IS GOOD” } is entered as the hoped behavior and \textit{convToUp} is retrieved.

SubLib21 is another library with type: double double → double. Some of the functions in this library are pure functions, some of them have side effects. Figure 4.10 shows some of components in subLib21.
Figure 4.10  Some Components in subLib21

For example: *add* and *addModi* are two functions, both of them add two double numbers and return the sum. But in *add*, two arguments' values remain the same after it is executed. But in *addModi*, the sum will also be stored in first argument after executing. This difference will cause the different behaviors of the two functions, and they will not be retrieved at same time. (See figure 4.11)

There are two functions, *min* and *mode*, in subLib21 have the same behaviors for the first input data, so them be treated as they are same, and grouped together. But using adapt skill, user can test if or not they are same. If not, one more input data is added into libLib21 for distinguishing them. (See figure 4.12, *min* and *mode* belongs to different groups)
Figure 4.11 Retrieval in subLib21

Figure 4.12 Some components in subLib21 After Adapting
4.2 Evaluation of the Approach and Implementation

Traditionally, retrieval performance is measured by recall and precision. Recall is defined as the ratio of the number of retrieved and relevant items to the total number of relevant items in the library. Precision is defined as the ratio of the number of retrieved and relevant items to the number of retrieved items.

From these definitions, we know that the idea condition is 100% recall, which means no more relevant item is missing, and 100% precision, which means all retrieved items are relevant items. But this evaluating assumes that any two components either relevant or irrelevant. So it is very hard to measure the relationship among the C functions. But we could discuss on general concepts.

In *Behavior Sampling* approach \(^{[31]}\), random test data is used to generate input that needlessly reduces the precision. Because the library manager is usually a much better source of input that distinguishes similar candidates. ORES approach overcomes the shortcomings by letting the library manager supply the inputs to improve the precision.

In OERS approach, we use a text file to classify the whole library into different sub-library. When user searches the candidate component, he/she first checks the file, then directly go to the correct one. This will greatly cut off the irrelevant components.

The main purpose of the OERS approach is to improve the efficiency of the execution-based retrieval. The retrieval time is used as the criterion.

Suppose total number of components in the library is N, The average of the execution time is \( T_1 \), then the retrieval time in general execution-based retrieval (GER) for one input test data is \( NT_1 \).

All functions in OERS are stored as a tree, retrieval becomes the searching
process among the tree. Usually the average test data needed to distinguish the functions is \(3^{[31]}\), which means the average depth of the tree is 3. In the worst case, the tree is like:

![Tree Diagram](image)

If \(T_2\) is the time to search one node, then the average retrieval time is \((1+2+3(N-2))T_2/N\).

The ratio compared with GER is:

\[
\text{Ratio} = \frac{N^2T_1}{(3(N-1))T_2}
\]

When \(N\) becomes big enough,

\[
\text{Ratio} = \frac{NT_1}{3T_2}
\]

The ratio is direct proportion to \(N\) because \(T_1/T_2\) can be treated as a constant. That means the more components in the library, the more benefit gain by using OERS approach.

In my experiment, three groups of components total 35 C functions are used to built the library.

- subLib11: \(\text{int} \rightarrow \text{long}\)
- subLib12: \(\text{char pointer} \rightarrow \text{void}\)
- subLib21: \(\text{double double} \rightarrow \text{double}\)

All functions in subLib11 are pure functions. All functions in subLib12 are procedures. The third group includes the functions with side effect.

We built reuse library, then retrieving each function from the library. Each retrieval time on the Server side was recorded. Then average retrieval time was computed. Another program, which implemented general execution-based retrieval, was
used to record the retrieval time by actually executing all the functions in each retrieval process. Table 4.3 lists the retrieval times by OERS approach and by GER approach in these three groups. The average benefit using OERS is larger than 7.

<table>
<thead>
<tr>
<th>sub-library name</th>
<th>retrieval time by using OERS (milliseconds)</th>
<th>retrieval time by using GER (milliseconds)</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>subLib11</td>
<td>192</td>
<td>1489</td>
<td>7.76</td>
</tr>
<tr>
<td>subLib12</td>
<td>150</td>
<td>1154</td>
<td>7.69</td>
</tr>
<tr>
<td>subLib21</td>
<td>156</td>
<td>1155</td>
<td>7.40</td>
</tr>
<tr>
<td>Average:</td>
<td>166</td>
<td>1266</td>
<td>7.63</td>
</tr>
</tbody>
</table>

Table 4.3 Compare Retrieval Time

4.3 Conclusion

In the thesis, we showed an approach for organizing imperative program components into a library for supporting execution-based retrieval. Based on the method, a prototype system called OERS is developed. This system could be used to construct and maintain an organized reuse library by providing the initialization, deletion, adaptation, and browse operations. A retrieval sub-system allows users to retrieve, test and view components.

OERS system provides efficient retrieval methods in terms of the retrieval time. The benefit is direct proportional to the total number of components in the library. It is 7 times faster in average than GER approach in our test. Once the library is well organized, users could retrieve more precise candidate component than in GER approach. OERS uses classification schema to cut off irrelevant components before searching in the library. This could improve the recall of retrieval.
OERS system is implemented by OOD and OOP. It is easy to modify, maintain, reuse and port to other computing environment. All the interfaces in OERS system are graphic, user-friendly, and easy-to-use.

Of course there are some limitations in the OERS system.

First, how to select the input data is a big problem. When initializing the library, we don’t know what data best suits for all the components. The depth of the tree may become very deep with increase of number of components. The depth could be N-1 if the total number of components is N in the worst case. The retrieval time could become very long. For this case, we need to reconstruct the library by trying other input data.

The second problem is the run-time problem: non-termination. We considered this problem in design. We are planning simply to set a timeout interval when running a function on an input values, terminating the computation after a fixed interval has passed. The method can be carried out by using multiple threads, but we did not implement it at here.

Besides above two problems needed to be solved, future work also includes other aspects. For examples: adding one more module to verify the input data provided by users; posting this approach to Internet so anybody can access the system through Internet browser; extending this approach to organized other executable components such as OO program components. This approach cannot be used to retrieve functions such as encryption functions, because we usually do not know the behavior of them for the input data in advance.
References


[3] Steven Atkinson, Roger Duke, " A Methodology for Behavioural Retrieval from Class Libraries" Software verification research center Department of computer science, the university of Queensland, Australia, technical Report, No. 94-28


[12] Young Park, "Organizing Reusable Components for Execution-based Retrieval" proceeding of CSACC '96 (International symposium of applied corporate computing)


Vita Auctoris

*June Xuejun Qiang* was born in WuXi, China. She obtained a Bachelor's degree in Electrical Engineering from *The Northwest Telecommunication Engineering Institute*, Xian, PRC in 1984. She received a Master's degree in Electric Engineering from *The Research Institute of Radio Measurement*, Beijing, PRC in 1987. She is currently a candidate for a Master's degree in Computer Science at *University of Windor* and expect to graduate in the fall of 1999.