Software component retrieval by composition.

Zhengyu Leo Wu

University of Windsor

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

Recommended Citation

https://scholar.uwindsor.ca/etd/3508

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. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

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.
Software Component
Retrieval by Composition

by
Zhengyu Leo Wu

A Thesis
Submitted to the Faculty of Graduate Studies and Research
through the School of Computer Science in Partial
Fulfillment of the Requirements for the Degree of
Master of Science at the
University of Windsor
Windsor, Ontario, Canada
1997
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.

L’auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L’auteur conserve la propriété du droit d’auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-31009-4
Abstract

Retrieving software component by composition helps user to build a new component using the existing components in the reuse component library when no single component in the component library can match the user requirements.

This thesis proposes a method of retrieval-by-composition in which user specifies the composition structure and the system retrieves all possible compositions that fit into the structure specified by the user. In our method, semantic properties, including demand, transmission and length properties and used to reduce the composing and retrieval space. The semantic properties of internal components are inferred based on the user requirements. A prototype system called WISER is implemented, which helps user visually compose the composition structure, then does the inference on each composition component and displays all possible compositions. The implementation design thoughts and details are discussed with actual examples of retrieval-by-composition and many screen shots. The study shows that using semantic properties has greatly reduced the final search space in an inexpensive and effective way.
To Fengming, Yiping and Yang
Acknowledgments

My deep gratefulness is given to my supervisor Dr. Y. G. Park for his patience, advice and guidance throughout this thesis work. This thesis work is a continuation of research work carried out by Dr. Park and his previous graduate students D. Ramjsingh, P. Bai and K Khalil. Their contribution has laid foundation for my work. Special thanks go to P. Bai and K. Khalil who helped me jump-start this version of WISER project and answered many of my early ignorant questions.

I am also deeply indebted to Dr. R. A. Frost for his important comments and advice and to Dr. C. S. Wong of Department of Mathematics and Statistics for many eye-opening discussions with him.
TABLE OF CONTENTS

Abstract ........................................................................ iv
Acknowledgments ......................................................... vi
List of Figures .............................................................. ix

1 Introduction .............................................................. 1
   Background on Software Reuse .................................... 1
   Component Retrieval for Reuse ................................. 3
   Component Retrieval by Composition ....................... 7
   Our Motivation and Approach ................................... 10

2 Composition Using Semantic Properties ................... 13
   Semantic Properties ................................................ 13
   Using Semantic Properties ....................................... 14
   Argument Demand Property Inference for Components .... 18
   Element Demand Property Inference for Components ...... 20
   Transmission Property Inference for Components .......... 24
   Length Property Inference for Components ................ 30

3 System Implementation of Retrieval–by–Composition ...... 37
   Overview ............................................................. 37
   Development Environment ...................................... 38
   System Structure .................................................. 40
   Internal Representation of Composition .................... 57

4 Retrieving Components Using the Prototype System ...... 62
   Step 1. Start the WISER Server ................................ 62
   Step 2. Connect as a client ....................................... 63
   Step 3. Retrieving single component ......................... 65
List of Figures

Figure 1       Mili and Mili's Composition Pattern A ............... 9
Figure 2       Mili and Mili's Composition Pattern B ............... 9
Figure 3       Mili and Mili's Composition Pattern C ............... 10
Figure 4       Mili and Mili's Composition Pattern D ............... 10
Figure 5       Basic Form of Composition: Single Path ............. 15
Figure 6       Basic Form of Composition: Parallel Paths .......... 15
Figure 7       Composing simple prinedoms constructor .......... 17
Figure 8       Simple Palindrome Constructor .................. 23
Figure 9       WISER Server Application Main Window ............. 42
Figure 10      Server-client message exchange illustration ....... 43
Figure 11      Server Class Collaboration Diagram ............... 45
Figure 12      Server Window After Service Thread Has Started 46
Figure 13      View Component Content. ....................... 47
Figure 14      Add Component to WISER Library .................. 48
Figure 15      Client Class Collaboration Diagram ............... 51
Figure 16      WISERClient applet Running on Netscape .......... 52
Figure 17      Context of A Component (hd) ..................... 53
Figure 18      Multiple Clients Running on Java Appletviewers. 54
Figure 19      ComposeApplet Running on Netscape ............... 55
Figure 20      Unit Dialog Box ............................... 56
Figure 21      Composition Example 1 .......................... 59
Figure 22      Composition Example 2 .......................... 60
Figure 23      Example 3. Composition Example 3 ............... 61
Figure 24      Start WISER Server ............................ 63
| Figure 25 | Retrieval Dialog ....................................... 65 |
| Figure 26 | Input Information for Retrieving a Single Component ....................................... 66 |
| Figure 27 | No result is returned from single component retrieval for spc ....................................... 67 |
| Figure 28 | Start a Retrieval-by-Composition Session ....................................... 68 |
| Figure 29 | New Unit ............................................... 69 |
| Figure 30 | Add the Module to Composition ....................................... 70 |
| Figure 31 | Final Composition Canvas for spc ....................................... 71 |
| Figure 32 | Retrieval By Composition Result ....................................... 72 |
| Figure 33 | Step-By-Step Retrieval-By-Composition Result ....................................... 74 |
Chapter 1 Introduction

1.1 Background on Software Reuse

Software Reuse  Software-reuse is the process of implementing or updating software systems using existing software assets [NIS1994]. In other words, software-reuse is the application of reusable software assets to other software systems [NIS1994]. Software-reuse may occur within a software system, across similar software systems, or in widely different software systems. Conveyed with the definition of reuse is the implication that reuse is a perspective and a judgment, that it is a way of thinking as a developer plans and performs engineering and development tasks. Reuse in general refers to software components as well as other engineering and development products [KRU1992].

Industry has demonstrated [LIM1994] that reuse of software will provide a basis for dramatic improvements in quality and reliability, speed of delivery, and in long-term decreases in costs for software development and maintenance. Matsumoto estimated based on his research [MAT1995] that even with a less than 50% reuse rate, component-based software development leads to reliability improvement as much as 10 times that of development that is not component-based.

Reusable Software Library  Many organizations focus their reuse initiatives on a reuse library where members of the organization can both store reusable assets and retrieve these assets when they need them. RSLs use specialized methods for component classification, search, and retrieval. A considerable number of tools and mechanisms for supporting reuse activities in software development have been
proposed. They provide assistance either to application developers for retrieving [MAA1991] [SIN1993] [FUG1991] [HEL1991], understanding [BOL1988], customizing [FUG1991] [BOL1988] [FIT1988] and composing components in a software library, or to library managers (operators) to create [FIT1988], [GIR1992], organize [SIN1993] [PRI1991] [TIC1989] and reorganize [CAS1991] reusable development information in the software library.

A survey of application programmers [BEL1992], which was conducted to discover user needs and attitudes toward reuse, shows that users consider reuse worthwhile, but most of them (especially those without object-oriented experience) expect more from application generators or from tools for automatic programming than from reuse systems like browsers.

Most software retrieval systems usually retrieve a set of reusable candidates ranked by similarity to user requirements [BEL1992] [FUG1991] [HEL1991] [BOL1988] [FIT1988]. For example, Software-Reuse Information Clearinghouse (ReuseIC) [REU000] is one of the major reuse systems on the web. The system stores RESOLVE/Ada, RESOLVE/C++, plain Ada and plain C++ components, documents and much other reuse informations. ReuseIC uses keyword based retrieval methods. ReuseIC has a formal interface. Having looked at its HTML source code, one would realize that the web interface of current system ReuseIC was written in CGI (Common Gateway Interface) script. Users can browse over the libraries through web links or search the software components by keywords.

However, as in the case of ReuseIC and as in many other reuse library systems, it is not desirable for users to invest a great effort in selecting a component. RSL might pop up dozens of components names as the result of a retrieval. These
component names, many in abbreviation, seem to be meaningless to most users. According to an investigation [BEL1992] in most cases users do not completely analyze the list of retrieved candidates provided by reuse systems even if the highly ranked components are discarded by the system. The investigation [BEL1992] also shows that users assume the best suited components are the ones on the top of the list of candidates (for example, the first to the third). Thus, to select a component from the list, the user examines only the associated information for those first components. If none of them satisfies the requirements, maybe the user will try to refine or rewrite the original query but, in most cases the user will simply abandon the search.

Girardi and Ibrahim concluded [GIR1993] that retrieval systems should exhibit more precision in their answers. It should discard some obviously unwanted components from the set of candidates and retrieve only the ones that satisfy more precisely the requirements of the user [GIR1993]. This demands better component retrieval mechanisms.

1.2 Component Retrieval for Reuse

A software component is a computational entity [NIS1994]. It can be an object, module, or encapsulated collection of functionality and services. Based on different specifications, information retrieval techniques have been frequently used to construct and evaluate software retrieval systems. People have explored various mechanisms of retrieving components. Major approaches include, keyword/facet based, formal specification based and semantic based approaches.

**Keyword-based and Facet-based Retrieval**  Following the research in that area, earlier software-reuse systems were keyword-based, such as in WAIS. In
these systems, catalogues are usually constructed and components are retrieved by specifying a set of keywords under a software classification scheme, using either a free vocabulary or a controlled one.

Traditional keyword based retrieval systems are limited by the expertise of users as well as by the size of the software libraries. It is often not desirable for users to find out proper keywords before any actual search. Moreover, the effectiveness of these systems is limited by the so-called "keyword barrier" [MAU1991], that is the systems are unable to improve retrieval effectiveness beyond a certain performance threshold. This situation is particularly critical in software retrieval where users require high precision and expect to retrieve only the best components for reuse without selecting from a list containing many irrelevant components. The availability of machine-readable dictionaries and techniques for language analysis have made possible the application of natural-language processing techniques to information retrieval systems [DAN1995] [SAL1989] [JAC1990].

Early in 1988 Wood and Sommerville [WOO1988] proposed a frame-based software component catalogue that tried to capture the meaning of software components. This catalogue is composed of a frame structure based around the action and for each basic function that the software performs. Frames have different slots for the different objects manipulated by the corresponding component. The retrieval system presents the frames as templates to be filled in by the user through the system menus. The frames in [WOO1988] can not parse input sentences. Users have to do the time-consuming search through the possible fillers. The indexing task, i.e., the construction of frames for the software component catalogue has to be performed manually.
Later Prieto-Diaz [PRI1991] proposed a multi-faceted classification scheme for cataloguing software components. This scheme has also been adopted by other reuse proposals, such as the REBOOT project [SIN1993]. The scheme uses facets as descriptors of software components. The scheme consists of four facets: the function performed by the component, the object manipulated by the function, the data structure where the function takes place (medium or location), and the system to which the function belongs [PRI1991]. The first three facets are considered sufficient by Prieto-Diaz [PRI1991] to describe the functionality of the component. For example (an example from me), the following terms describe a UNIX component: "locate" (function facet) "line numbers" (objects facet) in a "file" (data structure facet) are part of a "line editor" (system facet).

**Semantic Properties** Podgurski and Pierce [POD1993] have summarized approaches of component retrieval as two basic methods, which are syntactic retrieval and semantic retrieval. Semantic retrieval also includes methods that are formal-specification based and sampling-behavior based. Rittri suggested a third semantic-retrieval method that was type based [RIT1990].

Users are most-likely interested in finding a component that provides a particular service, that is, in the semantics of the components. But normally, neither a component’s name, nor the number of its parameters can be used to identify the semantics. Due to the lack of any distinguishing features, key phrases may be used to categorize the components. If this additional information is still not enough, fuzzy logic or neural networks, or other heuristic approaches are applied to identify proper artifacts of a collection.

*Type-based* retrieval uses data types of arguments and result as search key [RAM1994]. Researchers in the University of Windsor have been conducting
retrieval by other semantic properties during recent years. Several semantic properties have been proposed. These semantic properties are argument demand, element demand, transmission and length [PAR1995]. They are semantic properties of a component argument. The return result of component do not have these properties.

**Argument Demand** is how often an argument is needed. Possible values for argument demand property is "Always", "Sometime" and "Never". In many cases, an argument is always demanded to evaluate the component and generate the return. In some cases, an argument can simply be there for compatibility reason and not involved in any computation, thus argument demand property of this argument can be "Never". In the following example, the argument demand property argument y is "Sometime", because if \( x > 0 \), then y is not evaluated and if \( x \leq 0 \) then y is evaluated. As you might have thought, this semantic property only exists in languages which support lazy evaluation. For sake of research, we choose our library components all in Miranda functional language.

\[
sample\text{\_component} \ x \ y = 0, \text{ if } x > 0 \\
= y, \text{ otherwise}
\]

**Element Demand** property is another demand (evaluation) property. If the argument is not that of a simple type but has a structure, the demand property of its elements is Element Demand property. The possible values of Element Demand are "Always", "Sometime", "Never".

**Transmission** Property is about how many elements of an argument structure stay in the final return result. The possible values are "All", "Some", "None". For example, in function sort \( x \), transmission property of list argument \( x \) is "All" because all of its elements will be part of return result, although in a different
order (sorted order).

*Length* property is to compare the size of return result and a argument. Possible values are ">", "\geq", "\leq", "\leq", "\leq". For details of these semantic properties, please refer to [PAR1995]. Most recently retrieval by execution sampling was discussed in [PAR1996].

The use of a set of compatible design concepts can provide both a semantic framework for the catalog and guidelines for the internal structure of the components. A small, uniform, set of design concepts helps shorten the time needed to design, understand, and adapt the software components [JAZ1996]. The concepts impose certain standards on component interfaces and implementations. They provide a high-level framework for a contract between the provider of the component and its user. An appropriate set of design concepts should also guide the design of the components, and enable the easy adaptation of a component without changing the code. It should also establish the rules for component composition. The design considerations and implementation details will be given in Chapter 3 and Chapter 4.

### 1.3 Component Retrieval by Composition

When no existing library component can be found to satisfy user requirement, user may want to compose of a new component using existing ones. The requirements for a module can be decomposed into requirements for a set of sub-modules and patterns interacting between sub-modules. Formal methods use various reduction and factoring mechanisms to decompose specifications [CAR1993][CHE1993]. For nontrivial requirements, virtually infinite number of solutions of equal quality can be found. In this field, two different approaches have been proposed.
Hall's approach

Hall's approach is a black box approach [HAL1993]. It seems unthinkable at first to enumerate compositions of components within a library until one is found that has the desired behavior. Hall introduced a number of practical and theoretical considerations to make the number of composition manageable.

In Hall's approach, when no single component satisfies requirement, all possible two-level-deep compositions are constructed. If still nothing satisfies requirement, all possible three-level-deep compositions are constructed, and so on [HAL1993]. In general, the relation between the number of compositions and the composition depth is double-exponential.

Hall first used type information to limit the number of compositions. User then provided some input–output pairs. The compositions were tested by comparing the execution result with user provided input-output pair. This method was actually first proposed by Podgurski and Pierce [POD1993]. Hall extended it into compositions. More dramatic elimination in computing complexity was achieved by so-called dynamic programming. When generating compositions of depth \(d\), apply new components to all the compositions of distinct return value instead of distinct component of depth \(d - 1\).

Hall tested his algorithm on a library of 161 Lisp functions. Hall's prototype system was also written in Common Lisp. He limited the depth of compositions to three and level three components were limited to one-argument components. Running on a SUN SPARC II, fifteen queries took an average of 20 seconds and a maximum of 40 seconds. In one example, a query provoked 2400 component execution instead of a potential \(10^6\) executions. Thus this method is not entirely impractical.
However, Hall’s approach has introduced overall too much complexity in computation. In addition, it is very costly to verify the composition by actually executing the composition. It takes a lot of resource to do so. Verifying by execution is probably the last brute weapon we want to use.

**Mili and Mili’s approach**

Mili and Mili proposed a white box approach. Instead of Hall’s try-them-all strategy, four composition patterns were proposed as typical compositions. These four patterns are shown in Figures 1 to 4.

*Figure 1  Mili and Mili’s Composition Pattern A*

![Image of Pattern A](image)

*Figure 2  Mili and Mili’s Composition Pattern B*

![Image of Pattern B](image)
A composition is expressed either as one of above pattern or as concatenation of above patterns. However the completeness of these four patterns has been questioned by us. Using rigid patterns also forbids the scaling of system. Even Mili and Mili admitted their limitation. They stated [MIL1995] that there had not been even a heuristic to show that the composition can be completed and solved using their method.

1.4 Our Motivation and Approach

It is somehow apparent, as stated by some researchers[STE1995], a highly reusable program or library should bear the features of
a. generality: an algorithm works on all reasonable data types correctly;
b. efficiency: the speed of algorithm should not decrease due to the generality. Neither Hall’s nor Mili and Mili’s approach has met both above requirements. Hall’s method is not efficient. Mili and Mili’s method have provided no credible proof for using their four patterns. We acknowledge that composition of a new bigger component is not an easy task. thus we started by following the methodology of *incremental development.*

Assumption: *the user knows better what he wants than anybody else does.* It is rational to let the user construct the composition structure. System will thereafter infer the semantic properties of each internal based on the semantic properties of the out-most component and based on link structure between components. Then system will perform the retrieval. Instead of the dealing with difficult tasks of constructing composition, inferring semantic properties and retrieving by composition, we now focus only on semantic property inference and retrieval. The retrieval algorithm will be basically the same as that used in our previous research [PAR1995]. We also have designed an implementation that assists user visually compose the structure.

The rest of the Thesis will be organized as follows.

Semantic inference will be described in Chapter 2. This chapter contains most of algorithm and model that will form the core theory of this thesis and guidance for implementation.

For future developers and those interested in the prototype details, Chapter 3 gives the physical, logical and object designs of prototype implementation. The client-server model on which the prototype based-on will also be detailed in Chapter 3. Some screen shots have been taken to visually demonstrate the implementation details.
In Chapter 4, examples of retrieval-by-composition are given. The reader may also get a quick idea of how to use the prototype. The concepts of retrieval by composition is visually shown through examples step by step again.
Chapter 2  Composition Using Semantic Properties

2.1 Semantic Properties

The key of retrieval by composition is the issue of semantic inference of a composite. In other words, the problem is

What is properties of the return if we know input argument properties?

This thesis work, in addition to data type, adopts three more semantic properties as proposed in previous research conducted by Dr. Y. G. Park and his graduate student Mr. K. Khalil [KHA1995]. These three semantic properties are demand property, transmission property and length property.

Demand Property  Demand property of an argument indicates if the argument is has to be evaluated (i.e. demanded in evaluation) to determine the function result. The value of this property can be Never, Maybe and Always. For example, in the function (Miranda syntax),

\[ f \ x \ y = \begin{cases} x, & \text{if } y > 0 \\ 0, & \text{otherwise} \end{cases} \]

the Demand property of argument x is Maybe, because it might be evaluated in the case of \( y > 0 \). The Demand property of y is Always. Both are rather straightforward.

Demand property can also be applied to the elements of a complex argument. For example, the function `concatList` will append a list to the tail of another list.

\[
\begin{align*}
[] ++ ys &= ys \\
(x:xs) ++ ys &= x : (xs ++ ys)
\end{align*}
\]
The elements of both lists do not have to be evaluated at all to complete the function. So the Demand properties of elements of both arguments xs and ys are Never.

**Transmission Property** Transmission property can be applied to the elements of a complex argument. It indicates how many of the elements of an argument will be transmitted through the function and appear in the function return. The values of this property can be All, Some and None. The value of All stands for that all elements are transmitted and appear in the function return. The value of Some reflects that some elements are transmitted and appear in the function return. The value of None means no elements are transmitted and appear in the function return.

For the example of concatlist used in last section. All elements of xs and ys are transmitted and become part of the function return. No element is “lost” during function. Thus the transmission property value of both xs and ys are All.

**Length Property** Length property is only applicable to a complex argument. This property represents the relation between the size of an argument and the size of return. Use the example of concatlist again. Because the return list combines both input lists, the size of return list is no less than the size of either argument. Thus the length property of xs is “\( \geq \)”. The equal-to case happens when ys is an empty list. Similar with ys.

### 2.2 Using Semantic Properties

**Two Composing Paths** After carefully studied composition structures, we found that compositions can be represented as unidirectional graphs. On each node sits
a component. A path links two and only two component arguments. Because the semantic properties actually apply to arguments instead of component, it is logical to consider the composition structure as a set of unidirectional paths of arguments. The detailed graph study might help us understand better in composition, but due to the limitation of time frame and of my personal knowledge, this was not studied.

However, we found that two basic formats of paths can be concluded. Two path formats are Single Path and Parallel Path.

**Figure 5** Basic Form of Composition: Single Path

![Single Path Diagram]

**Figure 6** Basic Form of Composition: Parallel Paths

![Parallel Paths Diagram]

In the above two figures, each block represents a component which in turn can a single path or parallel paths. Using these two patterns, basically we can compose any composition structure.
Simple Observations  Before we start to do semantic inference, let us look at some simple intuitive observations first. Given a big module, which is the outermost component and which can be composed of some components and links between the arguments of these components, following observations hold true.

- No logical inference is needed for data type property. Direct-mapping should be applied to the types on two sides of a link
- The semantic properties of an outer module is given by the user.
- Splits are fully transparent to all semantic properties.
- Inference on semantic properties is the inference on components.

Example: A simple palindrome constructor

A simple palindrome constructor \textit{spc}, which stands for simple palindrome constructor, is going to be examined as an example for retrieval by composition. The function of \textit{spc} takes a list as input and generates a palindrome list by first sort the list then concatenate the list sorted list with its reversed list.

Assume the semantic properties of \textit{spc} are:

- data type: [a] \rightarrow [a]
- argument demand: ALWAYS
- element demand: ALWAYS
- transmission: ALL
- length: \geq

This component is not in the library and we try to compose it. The target module takes returns a palindrome list whose members are from the input list.

Assume the user gives this composition structure as below,
Component C1 is a one-argument component. Component C2 is also a one-argument component and component C3 is a two-argument component. In WISER library, the test library we use for research purpose, there are 35 one-argument components and 38 two-argument components. If no semantic information is considered, number of all possible combinations are:

$$35 \times 35 \times 38 = 46546.$$  

Now we are going to first consider data type information. As stated in above simple observations, argument data type does not needs inference, direct mapping is enough. Type Property of Module spc: [a]->[a]

Using direct data type mapping, C1 could possibly be

1. [a] -> [a] (6 possible components)
2. [a] -> a (2 possible components)
3. [a] -> num (2 possible components)

C2 could be

1. [a] -> [a] (6 components)
2. [a] -> a (2 components)
3. [a] -> num (2 components)
4. a->a (2 components)
5. num->num (12 components)

C3 could be
1. a -> [a] -> [a] (3 components)
2. [a] -> [a] -> [a] (3 components)
3. num->[a]->[a] (1 component)

Thus, after we have considered data type property, the number of compositions becomes much less:

10 (for C1) x 22 (for C2) x 7 (for C3) = 1540

Next we will discuss the inference of composing component semantic properties.

2.3 Argument Demand Property Inference for Components

Because the Demand property behaves differently when being applied to arguments and elements of an argument, we will discuss Argument Demand property inference and Element Demand property inference separately. First we will discuss Argument Demand property inference.

Argument Demand Property Inference

Case 1. Simple Path

∀Ai. Ai.Demand ≥ Module.Demand

Aj.Demand = Module.Demand

where Ai is any component along the pipeline, Aj is the first non-transparent component. If component has no effect on the semantic properties on its arguments and their transmission properties are “ALL”, then this component is transparent.
Otherwise the component is non-transparent. Below we give some details of inference in case 1 simple path situation.

**Module.ArgumentDemand = ALWAYS**  If the Argument Demand property of the module argument is ALWAYS, the Argument Demand property of each component argument along the path will be ALWAYS. In the case 1 situation,
1. C1.ArgumentDemand = ALWAYS
2. C2.ArgumentDemand = ALWAYS
3. C3.ArgumentDemand = ALWAYS

**Module.ArgumentDemand = MAYBE**  If the Argument Demand property of the module argument is MAYBE, the Argument Demand property of each component argument along the path will be MAYBE or ALWAYS, and there is at least one MAYBE along the path. In the case 1 situation,
1. C1.ArgumentDemand = MAYBE
2. C2.ArgumentDemand = ALWAYS/MAYBE
3. C3.ArgumentDemand = ALWAYS/MAYBE

If C1 is not a fully transparent component, then the Argument Demand of C1 must be MAYBE

**Module.ArgumentDemand = NEVER**  If the Argument Demand property of the module argument is NEVER, the Argument Demand property of each component argument along the path can be all values, and the first non-transparent component shall have the property value of NEVER. In the case 1 situation,
1. C1.ArgumentDemand = NEVER
2. C2.ArgumentDemand = ANY
3. C3.ArgumentDemand = ANY
If C1 is not a fully transparent component, its Argument Demand property should be NEVER.

**Case 2. Parallel Paths.** This is a logical OR case over the Case 1. As long as any one path of the parallel paths satisfies the condition the argument demand property of module will be satisfied. This is a disadvantage against the reduction of composition number.

### 2.4 Element Demand Property Inference for Components

**Case 1. Simple Path** In the case of simple path, there is at least one argument A_j has element demand value equal to the element demand of module argument. All arguments precede to A_j will have the Element Demand value of greater or equal to that of the module argument. There is no restriction to arguments which succeed A_j

\[ \exists A_j. Demand = Module.Demand \]
\[ \forall A_i \implies A_j. A_i. Demand \geq Module.Demand \]

where A_j is a component along the path. Below we give some details of inference in case 1 simple path situation.

**Module.ElementDemand = ALWAYS** If the Element Demand property of the module argument is ALWAYS, the first non-transparent component along the path must bear the Element Demand value of "ALWAYS". This is because if another argument along the path has Element Demand of "ALWAYS", the first element still has to be "ALWAYS" in Element Demand. In other word, Element Demand
property of component argument other than the first one along the path can be any value. In the case 1 situation,
1. C1.ElementDemand = ALWAYS
2. C2.ElementDemand = ALWAYS/MAYBE/NEVER
3. C3.ElementDemand = ALWAYS/MAYBE/NEVER

**Module.ElementDemand = MAYBE**  
Apply similar inference to the case where ElementDemand is “MAYBE”.
1. C1.ElementDemand = MAYBE
2. C2.ElementDemand = ALWAYS/MAYBE/NEVER
3. C3.ElementDemand = ALWAYS/MAYBE/NEVER
or
1. C1.ElementDemand = ALWAYS
2. C2.ElementDemand = MAYBE
3. C3.ElementDemand = ALWAYS/MAYBE/NEVER
or
1. C1.ElementDemand = ALWAYS
2. C2.ElementDemand = ALWAYS
3. C3.ElementDemand = MAYBE

In second possible composition above, Element Demand of second argument (C2) is “MAYBE” then the third argument (C3) can be any value. However, the first element (C1) can be “ALWAYS” and “MAYBE”, but the “MAYBE” case has already been included in the first composition, thus we reduced the second composition to the current form. Based on the same reasoning, we reduced the third composition to current form without the “MAYBE” case for first argument and second argument.
**Module.ElementDemand** = NEVER  NEVER is the smallest value of Element-Demand. There are three possible compositions.

1. C1.ElementDemand = NEVER
2. C2.ElementDemand = ALWAYS/MAYBE/NEVER
3. C3.ElementDemand = ALWAYS/MAYBE/NEVER
   
   or

1. C1.ElementDemand = ALWAYS/MAYBE/NEVER
2. C2.ElementDemand = NEVER
3. C3.ElementDemand = ALWAYS/MAYBE/NEVER
   
   or

1. C1.ElementDemand = ALWAYS/MAYBE/NEVER
2. C2.ElementDemand = ALWAYS/MAYBE/NEVER
3. C3.ElementDemand = NEVER

As long as one component argument has the ElementDemand value of "NEVER", the entire path can satisfy the ElementDemand property.

**Case 2. Parallel Paths.**  This is a logic OR case over the Case 1. Once the element property of any argument along any path of these parallel paths is greater or equal to that of module, the module element property will be satisfied. This character looses the restriction of ElementDemand property on components.

**Example**  Let us apply the argument demand and element demand inference to the sample module, the simple palindrome constructor spec.
Given, the module properties,

Argument Demand Property : ALWAYS

Element Demand Property : ALWAYS

Then, using the method described above, the Argument Demand properties of components can be inferred as below,

• C1 Argument1.ArgumentDemand = ALWAYS
• C2 Argument1.ArgumentDemand = ALWAYS
• C3 Argument1.ArgumentDemand = ALWAYS
• C3 Argument2.ArgumentDemand = ALWAYS

Next step, using the method described above, the Element Demand properties of components can be inferred as below,

Possible Composition 1

• C1 Argument1.ArgumentDemand = ALWAYS
• C2 Argument1.ArgumentDemand = ALWAYS or SOMETIME or NEVER
• C3 Argument1.ArgumentDemand = ALWAYS or SOMETIME or NEVER
• C3 Argument2.ArgumentDemand = ALWAYS or SOMETIME or NEVER
Possible Composition 2

- $C_1 \text{Argument1.ArgumentDemand} = \text{ALWAYS or SOMETIME or NEVER}$
- $C_2 \text{Argument1.ArgumentDemand} = \text{ALWAYS}$
- $C_3 \text{Argument1.ArgumentDemand} = \text{ALWAYS or SOMETIME or NEVER}$
- $C_3 \text{Argument2.ArgumentDemand} = \text{ALWAYS or SOMETIME or NEVER}$

Possible Composition 3

- $C_1 \text{Argument1.ArgumentDemand} = \text{ALWAYS or SOMETIME or NEVER}$
- $C_2 \text{Argument1.ArgumentDemand} = \text{ALWAYS or SOMETIME or NEVER}$
- $C_3 \text{Argument1.ArgumentDemand} = \text{ALWAYS}$
- $C_3 \text{Argument2.ArgumentDemand} = \text{ALWAYS or SOMETIME or NEVER}$

Possible Composition 4

- $C_1 \text{Argument1.ArgumentDemand} = \text{ALWAYS or SOMETIME or NEVER}$
- $C_2 \text{Argument1.ArgumentDemand} = \text{ALWAYS or SOMETIME or NEVER}$
- $C_3 \text{Argument1.ArgumentDemand} = \text{ALWAYS or SOMETIME or NEVER}$
- $C_3 \text{Argument2.ArgumentDemand} = \text{ALWAYS}$

Here, we have four possible compositions because of the two parallel paths which merge into $C_3$.

Finally we look into the WISER library. Apply the above inference to 1540 possible compositions left from type mapping. The number of possible compositions which also satisfy the both demand properties is 1480. We further decrease this number after transmission property inference.

### 2.5 Transmission Property Inference for Components

Because the transmission property behaves in an identical way when being applied to arguments and elements of an argument, we will discuss both cases together without discrimination.
Case 1. Simple Path. In case of simple single path, the transmission property of at least one argument is equal to that of module argument and all rest of arguments along path have the transmission property greater or equal to that of component argument.

\[ \forall A_i. A_i.\text{Transmission} \geq \text{Module.Transmission} \]

\[ \exists A_j. \text{Transmission} = \text{Module.Transmission} \]

where \( A_i \) is any component along the pipeline, \( A_j \) is any component. Above equations are obvious to some degree. They will be explained in detail as follow.

**Module.Transmission = ALL**  If all elements of the module arguments are transmitted to the final module return, then there can be no loss of element in anywhere in the path. All component argument along the path must have the transmission property of “ALL”, that is

1. \( C_1.\text{Transmission} = \text{ALL} \)
2. \( C_2.\text{Transmission} = \text{ALL} \)
3. \( C_3.\text{Transmission} = \text{ALL} \)

**Module.Transmission = SOME**  If only some elements of the module arguments are transmitted to the final module, then there can be no argument in the path has “NONE” transmission. Otherwise the result would be “NONE” for module argument. Further there must at least have one argument whose transmission value is “MAYBE”. All “ALL” along the path will lead to “ALL” for module component transmission value. Thus,

1. \( C_1.\text{Transmission} = \text{SOME} \)
2. \( C_2.\text{Transmission} = \text{ALL/SOME} \)
3. \( C_3.\text{Transmission} = \text{ALL/SOME} \)
or
1. C1.Transmission = ALL/SOME
2. C2.Transmission = SOME
3. C3.Transmission = ALL/SOME

or
1. C1.Transmission = ALL/SOME
2. C2.Transmission = ALL/SOME
3. C3.Transmission = SOME

The value “SOME” can appear at any component argument along the path.

**Module.Transmission = NONE**  In this case, one “NONE” along the path will be enough to generate the “NONE” transmission for the entire path. Three possible composition forms are,
1. C1.Transmission = NONE
2. C2.Transmission = ALL/SOME/NONE
3. C3.Transmission = ALL/SOME/NONE

or
1. C1.Transmission = ALL/SOME/NONE
2. C2.Transmission = NONE
3. C3.Transmission = ALL/SOME/NONE

or
1. C1.Transmission = ALL/SOME/NONE
2. C2.Transmission = ALL/SOME/NONE
3. C3.Transmission = NONE

**Case 2. Parallel Paths.** Parallel path case is a little complicated here. Two
situations can both satisfy the external transmission property requirement. In first situation, the maximum value of each path’s minimal transmission value equals the module argument’s transmission.

$$\max(\min_j(A_{i,j}.\text{Transmission})) = \text{Module.Transmission}$$

Or, in a second situation,

$$\max(\min_j(A_{i,j}.\text{Transmission})) < \text{Module.Transmission};$$

$$\sum_j \max(\min_j(A_{i,j}.\text{Transmission})) \geq \text{Module.Transmission}$$

In the first case, the elements which appear in the module return can be all transferred through the path whose minimal transmission is the biggest among all parallel paths. In the second situation, the elements which appear in the module return can be transmitted through various paths and merge into one path which finally lead to module return. Let us discuss in more details.

**Module.Transmission = ALL** Two equations for transmission property will generate the same result in this case. We will have three possible composition forms.

Possible Composition Form 1

1. C1.Transmission = ALL
2. C2.Transmission = ALL/SOME/NONE
3. C3, 1.Transmission = ALL
4. C3, 2.Transmission = ALL/SOME/NONE

Possible Composition Form 2

1. C1.Transmission = ALL/SOME/NONE
2. C2.Transmission = ALL
3. C3, 1.Transmission = ALL/SOME/NONE
4. C3, 2.Transmission = ALL
Possible Composition Form 3

1. C1.Transmission = SOME/ALL
2. C2.Transmission = SOME/ALL
3. C3.1.Transmission = SOME/ALL
4. C3.2.Transmission = SOME/ALL

**Module.Transmission = SOME**  Combine the result of two equations of transmission inference, we have

Possible Composition Form 1

1. C1.Transmission = SOME
2. C2.Transmission = ALL/SOME/NONE
3. C3.1.Transmission = SOME/ALL
4. C3.2.Transmission = NONE/SOME

Possible Composition Form 2

1. C1.Transmission = SOME
2. C2.Transmission = NONE/SOME
3. C3.1.Transmission = SOME/ALL
4. C3.2.Transmission = ALL/SOME/NONE

Possible Composition Form 3

1. C1.Transmission = SOME/ALL
2. C2.Transmission = ALL/SOME/NONE
3. C3.1.Transmission = SOME
4. C3.2.Transmission = NONE/SOME

Possible Composition Form 4

1. C1.Transmission = SOME/ALL
2. C2.Transmission = NONE/SOME
3. C3.1.Transmission = SOME
4. C3.2.Transmission = ALL/SOME/NONE

**Module.Transmission** = NONE

Possible Composition Form 1
1. C1.Transmission = NONE
2. C2.Transmission = ALL/SOME/NONE
3. C3.1.Transmission = NONE
4. C3.2.Transmission = ALL/SOME/NONE

Possible Composition Form 2
1. C1.Transmission = ALL/SOME/NONE
2. C2.Transmission = NONE
3. C3.1.Transmission = NONE
4. C3.2.Transmission = ALL/SOME/NONE

Possible Composition Form 3
1. C1.Transmission = ALL/SOME/NONE
2. C2.Transmission = NONE
3. C3.1.Transmission = ALL/SOME/NONE
4. C3.2.Transmission = NONE

Possible Composition Form 4
1. C1.Transmission = NONE
2. C2.Transmission = ALL/SOME/NONE
3. C3.1.Transmission = ALL/SOME/NONE
4. C3.2.Transmission = NONE

**Example**  Again we apply the above inference script on transmission property
to the simple palindrome constructor \( spc \).

\[
\begin{array}{c}
\text{C0} \\
\text{C1} \\
\text{C2} \\
\text{C3}
\end{array}
\]

\text{Module.Argument1.Transmission} = \text{ALL}.

Thus,

\text{C1.Argument1.Transmission} = \text{ALL} \\
\text{C2.Argument2.Transmission} = \text{ALL} \text{ or SOME or NONE} \\
\text{C3.Argument2.Transmission} = \text{ALL}

After looking into WISER library we conclude that number of possible compositions is reduced from 1492 to:

\[ 2 \text{ (for C1)} \times 10 \text{ (for C2)} \times 4 \text{ (for C3)} = 80. \]

\textbf{2.6 Length Property Inference for Components}

\textbf{Case 1. Simple Path}  The Length property is a little bit messy and less restricted comparing to semantic properties we discussed above. If some arguments’ length property values are greater than or equal to the module length value then there must be some other arguments’ length properties are smaller or equal to the module length value.
∃A_i.\text{Length} ≥ \text{Module.Length}:

∃A_j.\text{Length} ≤ \text{Module.Length}(i! = j)

\text{Module.Length} = \text{LT}

1. C_1.\text{Length} = \text{LT}
2. C_2.\text{Length} = \text{GT/GE/EQ/LE/LT}
3. C_3.\text{Length} = \text{GT/GE/EQ/LE/LT}

or

1. C_1.\text{Length} = \text{GT/GE/EQ/LE/LT}
2. C_2.\text{Length} = \text{LT}
3. C_3.\text{Length} = \text{GT/GE/EQ/LE/LT}

or

1. C_1.\text{Length} = \text{GT/GE/EQ/LE/LT}
2. C_2.\text{Length} = \text{GT/GE/EQ/LE/LT}
3. C_3.\text{Length} = \text{LT}

\text{Module.Length} = \text{LE}

1. C_1.\text{Length} = \text{LT/LE}
2. C_2.\text{Length} = \text{LE/EQ/GE/GT}
3. C_3.\text{Length} = \text{LT/LE/EQ/GE/GT}

or

1. C_1.\text{Length} = \text{LT/LE}
2. C_2.\text{Length} = \text{LT/LE/EQ/GE/GT}
3. C_3.\text{Length} = \text{LE/EQ/GE/GT}

or

1. C_1.\text{Length} = \text{LE/EQ/GE/GT}
2. C2.Length = LT/LE
3. C3.Length = LT/LE/EQ/GE/GT
   or
1. C1.Length = LT/LE/EQ/GE/GT
2. C2.Length = LT/LE
3. C3.Length = LE/EQ/GE/GT
   or
1. C1.Length = LT/LE/EQ/GE/GT
2. C2.Length = LE/EQ/GE/GT
3. C3.Length = LT/LE
   or
1. C1.Length = LE/EQ/GE/GT
2. C2.Length = LT/LE/EQ/GE/GT
3. C3.Length = LT/LE

**Module.Length = EQ**
1. C1.Length = LE/EQ
2. C2.Length = EQ
3. C3.Length = GE/EQ
   or
1. C1.Length = EQ
2. C2.Length = GE/EQ
3. C3.Length = LE/EQ
   or
1. C1.Length = EQ
2. C2.Length = LE/EQ
3. \[C_3.\text{Length} = \text{GE/EQ}\]

or

1. \[C_1.\text{Length} = \text{GE/EQ}\]
2. \[C_2.\text{Length} = \text{LE/EQ}\]
3. \[C_3.\text{Length} = \text{EQ}\]

or

1. \[C_1.\text{Length} = \text{LE/EQ}\]
2. \[C_2.\text{Length} = \text{GE/EQ}\]
3. \[C_3.\text{Length} = \text{EQ}\]

\textbf{Module.Length} = \text{GE}

1. \[C_1.\text{Length} = \text{GT/GE}\]
2. \[C_2.\text{Length} = \text{GE/EQ/LE/LT}\]
3. \[C_3.\text{Length} = \text{LT/LE/EQ/GE/GT}\]

or

1. \[C_1.\text{Length} = \text{GT/GE}\]
2. \[C_2.\text{Length} = \text{LT/LE/EQ/GE/GT}\]
3. \[C_3.\text{Length} = \text{GE/EQ/LE/LT}\]

or

1. \[C_1.\text{Length} = \text{GE/EQ/LE/LT}\]
2. \[C_2.\text{Length} = \text{GT/GE}\]
3. \[C_3.\text{Length} = \text{LT/LE/EQ/GE/GT}\]

or

1. \[C_1.\text{Length} = \text{LT/LE/EQ/GE/GT}\]
2. \[C_2.\text{Length} = \text{GT/GE}\]
3. C3.Length = GE/EQ/LE/LT
   or
1. C1.Length = LT/LE/EQ/GE/GT
2. C2.Length = GE/EQ/LE/LT
3. C3.Length = GT/GE
   or
1. C1.Length = GE/EQ/LE/LT
2. C2.Length = LT/LE/EQ/GE/GT
3. C3.Length = GT/GE

Module.Length = GT
1. C1.Length = GT
2. C2.Length = GT/GE/EQ/LE/LT
3. C3.Length = GT/GE/EQ/LE/LT
   or
1. C1.Length = GT/GE/EQ/LE/LT
2. C2.Length = GT
3. C3.Length = GT/GE/EQ/LE/LT
   or
1. C1.Length = GT/GE/EQ/LE/LT
2. C2.Length = GT/GE/EQ/LE/LT
3. C3.Length = GT

Case 2. Parallel Paths  Logic AND over Case 1.

Example  Again we apply above inference script to the simple palindrome constructor \( spc \).
Module. Argument1. Length = GE.

Having considered length property of 80 possible compositions left over by above inferences, the number is further reduced to 12.

And now because there are not very many candidates left over we can give all compositions below in the sequence of C1 — C2 — C3.

sort — listConcat — concatlist
sort — listConcat — merge
sort — reverse — concatlist
sort — reverse — merge
sort — sort — concatlist
sort — sort — merge
reverse — listConcat — concatlist
reverse — listConcat — merge
reverse — reverse — concatlist
reverse — reverse — merge
reverse — sort — concatlist
reverse — sort — merge
We finally reduced to 12 composition candidates from original 46546 possible compositions by considering type information, argument demand information, element demand information, transmission information and length information.
Chapter 3  System Implementation of Retrieval–by–Composition

Based on the semantic inference methods discussed in the previous chapter, we have developed a prototype system named WISER, which stands for WIndsor Software BasE of Reuse. This version of WISER system is based on the application developed by previous graduates students directed by Dr. Park and further equipped with a semantic inference engine. This chapter will give the details of how we have implemented this prototype.

3.1 Overview

The WISER prototype system consists of two parts — a component library and the application.

The component library stores the content of all components as well their indexed semantic properties including data type, argument demand, element demand, transmission and length property. The semantic properties of library components are stored in five index files, one property one file. The reason for keeping one property index in one file is that future modification, addition or deletion of a semantic property will only impact one file. This helps more object-oriented style programming.

Second part of WISER system is WISER application. The WISER application maintains, updates, retrieves and provides different views of the library components and their properties. If user can not find certain components in the library, and want to compose one, this version of WISER also help user visually compose a component and does the inference and show the possible compositions. The
single component retrieval is based on the methodology developed by Dr. Park and his previous graduate students.

The target environment running WISER prototype is intranet environment. The basic structure of this version of WISER application is client-server structure. WISER Server runs on server site and does all file I/O, library maintenance and updates. WISER Client can run anywhere on the intranet. Client does all user interactions, such as component composition and retrieval dialogs, visually and locally. It accesses component information by talking to remote server through the network. Client application does not change the WISER library directly. This provides better access control. The details will be described in following sections. First start with environment description.

3.2 Development Environment

Development Platform  The development of current WISER is done in two platforms, Sun Solaris 2.4 Unix and Microsoft Windows NT 4.0. The Unix operating system is installed on Scheofinkel machine of the School of Computer Science. The Microsoft NT is installed on PC. Final WISER has been successfully tested on Computer Science Scheofinkel system and an NT network with about 15 PCs connected by Novel LAN.

At the beginning of development, we thought that WISER retrieval-by-composition part can be implemented as a local application. However, considering the tremendous trend of intranet development, we believe put at least client function on the intranet would be nice. The work of putting WISER on web was started by Dr. Park and Ping Bai using CGI (Common Gateway Interface) tools during fall of 1995. In this version of WISER, server runs locally
on a Schoefinkel terminal and WISER clients can access WISER resources using any web browser with Java interpreter, such as Netscape (version 2.0 and up), Internet Explorer (3.0 and up), HotJava and et al.

Development Tools

We try to build the WISER implementation of retrieval by composition based on object-oriented philosophy. When this thesis work started, only two object-oriented development languages were accessible to students on Schoefinkel.

1. Sun C++ with XWindow/Motif APIs. XWindow and Motif API libraries has not been installed on Schoefinkel machine. But these libraries can be linked to programs at University of Windsor server SGI machine. Thus the final application has to run on SGI machine.

2. Sun Java. Java compiler has been downloaded and installed on Schoefinkel machine. Although there are basically no auxiliary development tools, many Java resources are free on the internet and easy to be downloaded from World Wide Web. The most important point is Java can embed its applications or, more precisely, so-called applets in HTML pages. This allows us to use the Java supported Web browser as basic window frame work.

Thus Sun-Java JDK (Java Developer's Kit) 1.02 was initially chose. Then after JDK 1.1 was released Java JDK, the WISER was modified and re-compiled using JDK 1.1.

Another advantage comes with using Java tool is I can easily embed the WISER client into a web page as an applet. The frame work of Web browser saves much time in window frame and graphic user interface development. Java also allows the cross-platform development and cross-platform execution. It is so-
called “Write once, run anywhere” style and this is an advantage in the intranet environment.

Java-enabled browsers such as Netscape, Hot Java and Internet Explorer provide simple and user-friendly methods to connect, search and browse the remote sites. Some researches found that with large organizations investing as much as 80 to 130 person-years to develop a formal software reuse library with its full-blown GUI [SIN1992], the WWW interface cost less than 1% of the cost to develop and maintain a standard, commercial-quality software reuse library [BER1994].

3.3 System Structure

The target of implementation is to write a set of programs which can physically test our ideas described in the the previous chapter. My philosophy guiding the entire implementation is that of object-oriented design and programming. The system starts with the basic user case scenarios.

User Cases  There are two categories of users and two basic user cases which I could think of. First kind of users are server operators and second kind of users are the client program users, or client users. Three kinds of the user cases are first, operator starts the server which waits for client connection; second, server operator does some library maintenance jobs. These two kinds jobs are in server application. Client users browse the components retrieve the components, compose the components and view their contents from WISER client application.
Client application and server application communicate with each other. The details are described in next subsection.

Client-Server structure
Above figure shows the main window of WISER Server application. The server service thread is statically created by the Server application as a worker thread when the server main application object is initialized. For security reasons, server thread does all the file I/Os. Clients download the semantic index information to their local memory and final retrieval of component context is done by message exchanges illustrated below.
To lessen the network work flow and achieve better performance potential, the implementation tries to keep as few message exchange between client-server as possible. All client user inputs are taken at the client site and pre-processed by the client application without knowledge of server. The client application sends the request message to the server only at the final context retrieval stage. Server then feedbacks the message including the requested component context or an error code. Appendix A2 lists all the server-client messages used by this version of WISER.
**Server Objects** The server user window contains some basic functions to run and maintain the service. It displays different semantic properties of the components. The semantic property lists are kept in overlapped list window. The reason to... The functions currently available as shown in buttons in above illustrations are.

1. **Start Server** — as described earlier this function starts the client service.
2. **Browse** — shows the lists of components with their semantic properties;
3. **Retrieve** — displays the context of the component;
4. **Delete** — delete the select component;
5. **Insert** — invoke a dialog box to add new component.

Following class diagrams show the important messages between classes. Due to limitation of space, two diagrams are used. One for server application and the other one for client application. Some classes, such as Index Set and Index Record, are duplicated classes in two diagrams.
The WISERServerWindow has been shown previously. When <Start Service> button is pressed, WISER server service thread will open the port 5188 and sits there waiting for client connection. The status label at the bottom of shows the port number and server status.
IndexFile class encapsulates all file read/write to local semantic index files. It inserts, deletes, packs, index records. IndexRecord is a record of semantic index consisting of a function name and semantic property values for each function argument.

MessageBox is used to display some message including the content of component. For example, when <Retrieve> button is pressed, a message box will appear to show the content of current selected component as shown below. The below figure show the content of component and
AddFunction class extends a dialogue and is used to append function to IndexFiles. To add a component, click <Insert> button in the Sever main window. A dialog box shown below will appear on screen. Then, just type in the component/function name and input the right semantic properties for this component.
Edit is not implemented in this version of WiserServer for the reason of data integrity. As you may have thought, it is probably better to implement WiserServer over some database tools than to write entire thing from scratch. If we utilize existing database tool, such as Oracle, we may take the advantage of its full blown features on data security and data integrity without stretch our own limits. I leave room for future changes on server side by separating server application from client application.

If click on <Delete> button on the main server window, the current selection will be deleted without a prompt. Here is another thing you may want to see a change: add a prompt dialog before proceed the final deletion. After the deletion is executed the program actually writes a deleted-mark, usually a character, to the component entry in the index file. The physical deletion of entire entry, that is the packing of the entry, happens at the exiting of the WiserServer program.
After completion of inputting one new component, click on <Save> to save the new component to library. If there are errors in the input, the program will pop up a message box informing you the details of errors. Notes that, this process only creates an entry of component in each semantic index file, the content file of the function has to be copied manually to the library folder. When you have finished adding all new component, click <Exit> to return to the main server window.

The client side of the application is a bit more complicated. Following figure show the class collaboration diagram of client classes. Again a list of files can be found in Appendix 2. Usually one class one file.

Click on the <Exit> button on the main server window, this will end the current server session. The WiserServer communicates with WiserClient objects through a telnet port. At any given time there can be only one program running on one local telnet port, so there can be only one copy of WiserServer running at any given time.

When the program is exiting, you can see some dump information about index file packing.

**Client Objects** The destination of client application is to allow client users to visually compose a set of components or, in a broader sense, units and retrieve these components or units.

The client side of WISER consists a set of Java applets embedding in html pages. For our own purpose of arguments, we still use the word 'application' here despite the conceptual difference between a full grown application and a Java applet. In term of coding, they are not clear gap between application programming and applet programming.
WISER is intranet targeted, so WISER client resides in a web page. Before starting the client, the server service should be started first. Running a WISER client without server service may cause unpredictable result.

The client-server messages listed in the previous section is coded in CommMessage class. CommMessage class is instantiated at both client and server side. It encapsulates all socket read/write operations and message parsing.

Start a copy Netscape or Java Appletviewer, then open the web page. The client session will start automatically within the frame work of Netscape or Java appletviewer. A demonstration of WISERClientApplet class is show below. The center portion of the screen is a window listing semantic properties of all components. The current visible semantic property can be changed by the radio buttons above the list. The functions are listed as buttons on top.
Figure 15 Client Class Collaboration Diagram
In the center, we can see a list of component semantic property values. This is a DataList. There are five DataList object on this applet, each holds a list of values of one semantic property. Only one DataList object is visible at a time. User can toggle through them by clicking on the radio buttons on the upper part of applet. As you might have guessed, the DataList objects actually store and display the index record information. IndexRecords are sent by WiserServerThread through CommMessage class which is socket I/O class.
When click on <Retrieve> button, a UnitDialog dialog box will appear. This dialog will be shown in a figure later when we explain the ComposeApplet class.

Click on the <View Context> button, the context of the current selected component will be ftp-ed and shown in a new page as displayed below.

To view list of different semantic properties, you may click on the radio buttons at the top of the window to switch between overlapped list windows. I displayed semantic properties separately because I believed it would be easier to add new properties to the list.

Figure 17 Context of A Component (hd)

At any given time there can be a number of clients running concurrently. In term of design, there is no limitation on maximum number of clients. However, in real world, the upper bound of number of concurrent clients allowed depends
on the machine resource. Although multiple clients can be running concurrently, at time point there is only one client allowed to exchange data with server and all other clients are waiting in a queue for their turn. Figure 5 shows a set of WISER client applets running concurrently.

Figure 18 Multiple Clients Running on Java Appletviewers.

The objects related to ComposeApplet will be explained in next sub-section. ComposeApplet the main object for Retrieval-By-Composition. When <Compose> button on WISERClientApplet is clicked compose.html, on which ComposeApplet.class is running, will be loaded into web browser.
Composition Objects  Next figure shows the ComposeApplet running on a Netscape 3.0 browser.

Figure 19  ComposeApplet Running on Netscape

Figure 9. Component Composition Applet

A UnitDialog is invoked by clicking on <Add Unit> or <Edit Unit> button on the applet. The dialog is illustrated below. If invoked by <Add Unit>, the caption of dialog will be “New Unit”. If invoked by <Edit Unit>, the caption of dialog will be “Edit Unit”. If invoked by <Retrieve> in WISERClientApplet
(last section), the caption of dialog will be "Retrieve Component". User can input or edit the semantic information of a component or unit. The difference between a component and a unit will be explained in next sub-section: internal representation of composition.

Figure 20  Unit Dialog Box

The composition can be visually completed using the buttons on ComposeApplet and mouse click-draggs. Then clicking on the <Retrieval By Composition> button will creates an inference engine to perform semantic inference based on the composition and semantic information of module (out-most) component provided by user. The InferenceEngine class is a base class for inference. But details of inference mechanism described in last chapter are implemented in PathInference class and CompositionInference class. Both of them are derived from InferenceEngine class. The result of inference and composition will display on the DataList on the bottom of the applet.
If user wants to view the step by step details of inference of composition, he uses the buttons at the lower part of applet besides the DataList. Clicking on radio buttons there will let inference engine infer component one property at a time.

3.4 Internal Representation of Composition

Internal Data Structure of Component Composition Above we have described the program structure of WISER. Now we are going to describe the internal representation of a composition. In order to describe a composition, three types of Units are used to represent the composition of a big component. Three types of units are,

1. Module — the out-most component is module. The semantic properties of module are supposed to be given by user. If user does not specify a semantic property value, the semantic inference of internal components will be not be conducted. For example, spc in Chapter 2 is a module

2. Component — the unit that will be retrieved from library later. For example, C1, C2, C3 in last chapter are components

3. Split — it split an argument of module or return of component to feed into more than one argument down in the path.

Units are linked to each other through links. Actually the link is the attribute of Argument objects. A unit may own some arguments and they form a composition by pointing the links of arguments one to another.

The next table describes all the legal links between various kind of composition units. There are some link types are forbidden. SPLIT can feed its outlinks
<table>
<thead>
<tr>
<th>COMPO.</th>
<th>SPLIT</th>
<th>MODULE</th>
</tr>
</thead>
<tbody>
<tr>
<td>X.R.Link = Y.Ai</td>
<td>X.R.Link = Y.R</td>
<td>X.R.Link = Y.R</td>
</tr>
<tr>
<td>Y.Ai.Link = X.R</td>
<td>Y.R.Link = X.R</td>
<td>Y.R.Link = X.R</td>
</tr>
<tr>
<td>X.R.IO = OUT</td>
<td>X.R.IO = OUT</td>
<td>X.R.IO = OUT</td>
</tr>
<tr>
<td>Y.Ai.IO = IN</td>
<td>Y.R.IO = IN</td>
<td>Y.R.IO = IN</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SPLIT</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>X.Ai.Link = Y.Aj</td>
<td>Ø</td>
<td>Ø</td>
</tr>
<tr>
<td>Y.Aj.Link = X.Ai</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X.Ai.IO = OUT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y.Aj.IO = IN</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MODULE</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>X.Ai.Link = Y.Aj</td>
<td>X.Ai.Link = Y.R</td>
<td>Ø</td>
</tr>
<tr>
<td>Y.Aj.Link = X.Ai</td>
<td>Y.R.Link = X.Ai</td>
<td></td>
</tr>
<tr>
<td>X.Ai.IO = OUT</td>
<td>X.Ai.IO = OUT</td>
<td></td>
</tr>
<tr>
<td>Y.Aj.IO = IN</td>
<td>Y.R.IO = IN</td>
<td></td>
</tr>
</tbody>
</table>

into components but not another split or the module. Module can not link to itself. These are rules for simple implementation. Rest of rules are quite self explanatory. A few highlights are given below,

1. A COMPONENT inputs from As and outputs to R
2. A SPLIT inputs from R and output to As
3. A MODULE inputs from R and outputs to As
4. There is ONE MODULE in any composition
5. A SPLIT ONLY outputs to COMPONENTs.
Now we are ready to give several examples to show how actually the compositions are expressed internally in the program.

**Example 1. A simple 2-component 1-argument pipeline case.** The example is visually shown below.

![Diagram](image)

**Figure 21** Composition Example 1

First we will declare all units.

- $C_0(A_1, R, \text{MODULE}, 1)$
- $C_1(A_1, R, \text{COMPONENT}, 1)$
- $C_2(A_1, R, \text{COMPONENT}, 1)$

Then we define links

- $C_0.A_1.\text{Link} = \neg C_1.A_1; C_0.A_1.\text{IO} = \text{OUT}$
- $C_1.A_1.\text{Link} = \neg C_0.A_1; C_1.A_1.\text{IO} = \text{IN}$
- $C_1.R.\text{Link} = \neg C_2.A_1; C_1.R.\text{IO} = \text{OUT}$
- $C_2.A_1.\text{Link} = \neg C_1.R; C_2.A_1.\text{IO} = \text{IN}$
- $C_2.R.\text{Link} = \neg C_0.R; C_2.R.\text{IO} = \text{OUT}$
- $C_0.R.\text{Link} = \neg C_2.R; C_0.R.\text{IO} = \text{IN}$

The compositions are expressed by unit declaration and link declaration above.
Example 2. A Two Pipeline Merge Case  The composition is illustrated below.
First, declare of units as

Figure 22 Composition Example 2

\[
\begin{array}{c}
\text{C0} \\
\text{C1} \\
\text{C2} \\
\text{C3} \\
\text{C0}
\end{array}
\]

C0(A1, A2, R, MODULE, 2)
C1(A1, R, COMPONENT, 1)
C2(A1, R, COMPONENT, 1)

Second, define links,
C0.A1.Link = !C1.A1; C0.A1.IO = OUT
C1.A1.Link = !C0.A1; C1.A1.IO = IN
C0.A2.Link = !C2.A1; C0.A2.IO = OUT
C1.R.Link = !C3.A1; C1.R.IO = OUT
C2.R.Link = !C3.A2; C2.R.IO = OUT
C3.A2.Link = !C2.R; C3.A2.IO = IN
C3.R.Link = !C0.R; C3.R.IO = OUT
C0.R.Link = !C3.R; C0.R.IO = IN
Example 3  Again the composition is expressed by units and links between arguments.

Figure 23 Example 3. Composition Example 3

Declaration of units

C0(A1, R, MODULE, 1)
C1(A1, R, COMPONENT, 1)
C1(A1, A2, R, SPLIT, 2)
C2(A1, A2, R, COMPONENT, 2)

Defining links.

C0.A1.Link = !C1.A1; C0.A1.IO = OUT
C1.A1.Link = !C0.A1; C1.A1.IO = IN
C1.R.Link = !C2.R; C1.R.IO = OUT
C2.R.Link = !C1.R; C2.R.IO = IN
C3.A2.Link = !C2.A2; C3.A2.IO = IN
C3.R.Link = !C0.R; C3.R.IO = OUT
C0.R.Link = !C2.R; C0.R.IO = IN
Chapter 4  Retrieving Components
Using the Prototype System

In this chapter, we will demonstrate how to use WISER prototype system to retrieve a component from library using retrieval-by-composition. The example we will use is the simple palindrome constructor we used in previous chapters. The details were given in page 13. We will do retrieval step by step.

4.1 Step 1. Start the WISER Server

If the WISER Server has already been started, then this step can be skipped.

Step 1.1  Under shell prompt, type

\textit{java WiserServer}

will start the server application. At any given time, only one copy of server program can run.

Step 1.2  Click on <Start Server> button.

The button is on the top of the window. This will start the server service thread waiting for client connection. Note that the status label changes after clicking the <Start Server> button. The label also shows the local port ID on which the server is running.
4.2 Step 2. Connect as a client

Start the Netscape 2.0 or higher (or Internet Explorer 3.0 or higher), open the wiser.html. This page shall show up. The applet loading may be slow.
<table>
<thead>
<tr>
<th>Function</th>
<th>Arguments</th>
<th>Return Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>nats</td>
<td>0</td>
<td>[num]</td>
</tr>
<tr>
<td>and</td>
<td>1 (bool)</td>
<td>bool</td>
</tr>
<tr>
<td>avgr</td>
<td>1 (num)</td>
<td>num</td>
</tr>
<tr>
<td>boolSum</td>
<td>1 (a)</td>
<td>[a]</td>
</tr>
<tr>
<td>divisors</td>
<td>1 (num)</td>
<td>[num]</td>
</tr>
<tr>
<td>elizero</td>
<td>1 (a)</td>
<td>[a]</td>
</tr>
<tr>
<td>even</td>
<td>1 (num)</td>
<td>bool</td>
</tr>
<tr>
<td>fact</td>
<td>1 (num)</td>
<td>num</td>
</tr>
<tr>
<td>fib</td>
<td>1 (num)</td>
<td>num</td>
</tr>
<tr>
<td>fst</td>
<td>1 (a)</td>
<td>a</td>
</tr>
<tr>
<td>hd</td>
<td>1 (a)</td>
<td>a</td>
</tr>
<tr>
<td>id</td>
<td>1 (a)</td>
<td>a</td>
</tr>
<tr>
<td>init</td>
<td>1 (a)</td>
<td>[a]</td>
</tr>
<tr>
<td>last</td>
<td>1 (a)</td>
<td>a</td>
</tr>
<tr>
<td>lenlist</td>
<td>1 (a)</td>
<td>num</td>
</tr>
<tr>
<td>listConcat</td>
<td>1 ([a])</td>
<td>[a]</td>
</tr>
<tr>
<td>lstdiv</td>
<td>1 (num)</td>
<td>num</td>
</tr>
<tr>
<td>max</td>
<td>1 (a)</td>
<td>a</td>
</tr>
<tr>
<td>middle</td>
<td>1 (a)</td>
<td>a</td>
</tr>
<tr>
<td>min</td>
<td>1 (a)</td>
<td>a</td>
</tr>
<tr>
<td>neg</td>
<td>1 (num)</td>
<td>num</td>
</tr>
<tr>
<td>negation</td>
<td>1 (bool)</td>
<td>bool</td>
</tr>
</tbody>
</table>
Now you are connected as a WISER client, and we can start to retrieve. First we will see if a single component will satisfy all our requirements.

4.3 Step 3. Retrieving single component

Step 3.1 Click on the <Retrieve> button on the above screen. The next dialog box will prompt you to input retrieval requirements.

Figure 25 Retrieval Dialog

Step 3.2 Type in all information you know about the component as follow, then click <OK>
Step 3.3 As we can see from the blank screen below, no result is return from library retrieval. So we head for composition.
4.4 Step 4. Ready For Composition

Click on the <Compose> button will start a retrieval-by-composition client session as shown below. The canvas (the central grid part) is now blank, we start to compose right away. If the canvas has been drawn before and we want to start a new session, we can click on the <ClearAll> button. As you might have noticed, there are six buttons on top of the screen for composing the internal structure. These buttons are <New Unit>, <Edit Unit>, <Delete Unit>, <New
Link>, <Edit Link> and <Clear All>. We will use these buttons and the mouse to compose the structure.

Figure 28 Start a Retrieval-by-Composition Session

\begin{center}
\begin{verbatim}
mat  0 argument(s).  \rightarrow [num]
end 1 argument(s).  [bool]  \rightarrow bool
wpr 1 argument(s).  [num]  \rightarrow num
bool2sum 1 argument(s).  [a]  \rightarrow [a]
\end{verbatim}
\end{center}

4.5 Step 5. Composing Structure

Step 5.1 First we should add a module to the canvas. Each composition must have one and only one module unit.
Click on the <New Unit>. The New Unit Dialog box will appear.

Figure 29 New Unit

First, choose the "Unit Type" as "Module". Then we input all semantic information of module. If some semantic information are stated as "Unknown", the system will ignore that property in future process. When input is finished, click on <OK> to finish.

The module will appear on screen. The module has a double line on its bottom-right, drag on there will resize the module.
Step 5.2  Next, we will start to add all the component units and splits to the canvas by repeating the same steps for adding the module. We do not have to specify any semantic property for components, the system will infer the semantic properties of each component for us. But if we do specify, then the system will not infer and will instead use the semantic values we input to retrieve. Unless
user knows exactly the information, it is not recommended for user to input component semantic property directly.

If in the middle of somewhere, we want to change the unit information, we can click on <Edit Unit>.

If we want to delete a unit, we can use <Delete Unit> button.

If we want to link two units, we click on <New Link> then drag the mouse from one end of the link to the other.

If we want to delete a link, we click on any end of the link to highlight to link, then click <Delete Link>

Next figure shows the final composition canvas for

Figure 31 Final Composition Canvas for spec
4.6 Step 6. Retrieval By Composition

Click on the <Click To Retrieve By Composition> button. The system will now start to infer and retrieve all possible compositions. The result will show in the list at the lower part of the applet.

Figure 32 Retrieval By Composition Result

<table>
<thead>
<tr>
<th>Sort</th>
<th>Reverse</th>
<th>Concatlist</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sort</td>
<td>Reverse</td>
<td>Merge</td>
</tr>
<tr>
<td>Sort</td>
<td>Sort</td>
<td>Concatlist</td>
</tr>
</tbody>
</table>

If user wants to see retrieval-by-composition details step by step, he can.
4.7 Step 7. Step by Step Retrieval-by-Composition

For example, we retrieve component #3.

**Step 7.1** Click on component 3 to highlight it. Then click on <Browse> button, this will list all the two-argument components in the library.

**Step 7.2** Click on the <Type> radio button to start with type inference first. Then click on retrieve, the type information will be inferred and all possible results will be listed.

**Step 7.3** Click on the <Argument Demand> radio button to do retrieval with argument demand inference. Then, click on <Retrieve> button. We can see the number of possible candidates have been reduced.

**Step 7.4** Repeat last step for Element Demand, Transmission and Length semantic properties. We will see the number of candidates reduce as we proceed.

**Step 7.5** We can get the result shown below. We see two candidates left.
So far we have briefly shown the WISER prototype system. Because the object-oriented design of prototype, the system has left room for future scaling and modification. Lots of GUI components are also reusable.
Chapter 5 Conclusion

Component Retrieval-by-Composition actually consists of three steps. First is to specify the structure, second user requirement verification and finally is the retrieval. By assuming the user know better about the structure, we let user provide the composition structure and our logic does the verification and retrieval.

In order to reduce the search space, we utilize semantic properties inference. In the example presented in the thesis, the simple palindrome constructor, given user provided composition structure, we had 46546 candidate compositions at first. After inference on semantic properties, such as data type, argument demand, element demand, transmission, and length, respectively, we reduced the search space step-by-step to only 12 candidate. It is very encouraging.

The implementation part of retrieval-by-composition is not complicated and can be reproduced quite easily. This provides a promising new methodology in software reuse.
## APPENDIX A.1 WISER File Lists

<table>
<thead>
<tr>
<th>File Name</th>
<th>Purpose</th>
<th>Used-in</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddFunction.java</td>
<td>Add a new component semantic index to WISER library.</td>
<td>server application</td>
</tr>
<tr>
<td>CommMessage.java</td>
<td>Encapsulates all socket read write and message parsing between server and client</td>
<td>server, client, composition</td>
</tr>
<tr>
<td>ComposeCanvas.java</td>
<td>The center canvas in ComposeApplet to visually compose and display composition structure.</td>
<td>composition</td>
</tr>
<tr>
<td>DataList.java</td>
<td>A list with information of index records.</td>
<td>server, client, composition</td>
</tr>
<tr>
<td>InferenceEngine.java</td>
<td>The base class for PathInference and CompositionInference</td>
<td>composition</td>
</tr>
<tr>
<td>MessageBox.java</td>
<td>A message box.</td>
<td>server, client, composition</td>
</tr>
<tr>
<td>Mouse.java</td>
<td>Mouse states</td>
<td>composition</td>
</tr>
<tr>
<td>PathInference.java</td>
<td>Inference engine for step by step infer and composition</td>
<td>composition</td>
</tr>
<tr>
<td>Class Name</td>
<td>Description</td>
<td>Location</td>
</tr>
<tr>
<td>----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>PossibleArgProperties.java</td>
<td>Part of Argument object listing all possible argument property values.</td>
<td>composition</td>
</tr>
<tr>
<td>SemanticProp.java</td>
<td>The string representation of all semantic property values</td>
<td>server, client, composition</td>
</tr>
<tr>
<td>Setup.java</td>
<td>The basic WISER system set-up information</td>
<td>server, client, composition</td>
</tr>
<tr>
<td>UnitDialog.java</td>
<td>The dialog box used to retrieve, add and edit unit or components</td>
<td>client, composition</td>
</tr>
<tr>
<td>UnitVector.java</td>
<td>A vector of Unit objects. It is used in composition to list all units in a composition structure.</td>
<td>composition</td>
</tr>
<tr>
<td>WiserClientApplet.java</td>
<td>The main client applet</td>
<td>client</td>
</tr>
<tr>
<td>WiserServer.java</td>
<td>The main server program</td>
<td>server</td>
</tr>
<tr>
<td>WiserServerThread.java</td>
<td>The server service class</td>
<td>server</td>
</tr>
<tr>
<td>Argument.java</td>
<td>An argument with semantic properties.</td>
<td>server, client, composition</td>
</tr>
<tr>
<td>Unit.java</td>
<td>A unit which has type (component, split or module), arguments and other attributes</td>
<td>client, composition</td>
</tr>
<tr>
<td>WiserPanel.java</td>
<td>A panel developed for convenience of use.</td>
<td>client</td>
</tr>
<tr>
<td>Class Name</td>
<td>Description</td>
<td>Use Case</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>IndexFile.java</td>
<td>The class encapsulates all file I/O to semantic index files.</td>
<td>server</td>
</tr>
<tr>
<td>IndexRecord.java</td>
<td>A record of values of one semantic property of a component.</td>
<td>server, client,</td>
</tr>
<tr>
<td></td>
<td>retrieve-by-composition</td>
<td>composition</td>
</tr>
<tr>
<td>ComposeApplet.java</td>
<td>The main applet for retrieve-by-composition</td>
<td>client.</td>
</tr>
<tr>
<td>SemanticSet.java</td>
<td>A set of all semantic properties, used only of inference intermediate purpose</td>
<td>composition</td>
</tr>
<tr>
<td>Composition.java</td>
<td>A possible composition of components for inference purpose.</td>
<td>composition</td>
</tr>
<tr>
<td>ComponentUnit.java</td>
<td>A component unit with some attributes to facilitate the composition inference</td>
<td>composition</td>
</tr>
<tr>
<td>CompositionInference.java</td>
<td>The main retrieval-by-composition inference engine. No step-by-step ability.</td>
<td>composition</td>
</tr>
<tr>
<td>ComponentVector.java</td>
<td>a vector for component units. used only as intermediate data for inference</td>
<td>composition</td>
</tr>
</tbody>
</table>
APPENDIX A.2 WISER Client-Server Messages

**Message Header Strings**
1. Hello : "hi!"
2. Bye : "bye"
3. Acknowledge : "ack"
4. Get : "get"
5. Retrieve : "ret"
6. Compose : "com"
7. Context : "con"
8. Start : "sta"
9. Line : "lin"
10. End : "end"
11. Put : "put"

**Special Content Strings**
1. DataType : "data type"
2. ArgDemand : "argument demand"
3. ElemDemand : "element demand"
4. Transmission : "transmission"
5. Length : "length"
BIBLIOGRAPHY


Laboratory, Gaithersburg, MD, December 1994.


VITA AUCTORIS

Mr. Zhengyu Leo Wu was born in 1970 in Jiangsu, China. He graduated from No. 85 High School, Xi’an, China in 1987. From there he went on to the Xi’an Jiaotong University at Xi’an, China where he obtained a Bachelor’s and a Master’s Degree in Electrical Engineering. He is currently a candidate for the Master’s degree in Computer Science at the University of Windsor and hopes to graduate in the Spring of 1997.