A Discretization Technique Based on Behavior for Normalizing Diversified Code Reuse Repositories

Swathy Vodithala
Assistant Professor, Dept. of CSE, KITS, Warangal
Suresh Pabboju
Professor, Dept. of IT, CBIT, Hyderabad

ABSTRACT*:* The major motivation of Component Based Software Development (CBSD) or Component Based Software Engineering (CBSE) is Software reuse. The importance of software reuse has been magnified as mostly the component reused refers to source code rather than documentation, tools and design patterns. The component in the proposed work is source code taken from the reuse repository. The proposed work explains a technique that can extract the behavior of software components taken from diversified code reuse repositories. The behavior of components taken from different code reuse repositories is normalized to a single dataset which can further be used by any retrieval algorithm. Apart from the behavior other relevant facets can be identified to describe the software component.

KEYWORDS: *Behavior; Component Description; Discretization; Diversified dataset; Code reuse.*

1. INTRODUCTION

Software engineering is the application of engineering to software. Component-Based Software Development (CBSD) or Component Based Software Engineering (CBSE) focuses on the development of applications based on existing software components rather than doing it from scratch.[2][3][4]. CBSE is having a higher level of importance as it is the key technology followed by many industries and resulting in high-quality software systems that are developed on time. The main aim of CBSE is to minimize the cost and time, consequently gives profitable results. The components are the pieces of code written by different programmers belong to different companies who follow different standards. The internal assumptions of code reuse repository differ because of different standards followed by the companies, so the substitutability and compatibility of software components plays a vital challenge in CBSD. The adaptability of a component is to be verified, because in reuse a component must successfully replace another in a particular application. .Reuse may be on design pattern, program elements or tools .But the widely reused software component is source code. There are three major areas in software engineering which has to be focused when considering the components for software reuse [5]. These are described as

a) Classifying/Clustering the components needed.

b) Describing the components.

c) Finding the appropriate component.

The structure of the paper is organized as follows. The section II, describes the related work which explains the existing approaches that forms the basis for proposed work. The section III explains the proposed algorithm along with the architecture .In the section IV, we have the experimentation results. The paper is concluded by conclusion, future scope and references.

2. RELATED WORK

There are many ways by which the behavior of a software component can be explained. The word specification can also be interchangeably used to the term behaviour[1]. The behavior of software is normally explained by one of the techniques given below:

- Informal specifications
  - a) comments embedded in code
  - b) informal metaphors
- Formal specifications
  - a) formal mathematics.
    - i) algebraic specifications
    - ii) model based specifications
  - b)predicate calculus

2.1. Specifications of components using Larch

There are many ideas proposed to retrieve a behavior of a software component. The specifications of software components have been compared by Zamarski both for functions (e.g., C routines, Ada procedures, ML functions) and modules (roughly speaking, sets of functions) written in some programming language [8]. These components might typically be stored in a program library, shared directory of files, or software repository. Associated with each component is a signature and a specification of its behavior .

Whereas signatures describe a component's type information (which is usually statically-checkable), specifications describe the component's dynamic behavior. Specifications more precisely characterize the semantics of a component than just its signature. The specifications are formal, i.e., written in a formally defined assertion language. Although we define match as a conjunction, we can think of signature match as a "filter" that eliminates the obvious non-matches before trying the more expensive specification match. The importance is to write pre-and post-condition specifications for each function, where assert ions are expressed in a first-order predicate logic. Match between two functions is then determined by some logical relationship, e.g., implication, between the two pre-/post-condition specifications. We can then define match between two modules in terms of some kind of match between corresponding functions in the modules. Given our choice of formal specifications, we can exploit state-of-the-art theorem proving technology as a way to implement a specification match engine.

The main disadvantage of specification matching is that it is very expensive because to extract the behavior meta language has to be defined .one more problem with specification matching is interoperability.

2.2 Behavioral matching by executing the components

Atkinson proposed behavioral retrieval which works by exploiting the executability of software components. Programs are executed using components, and the responses of components are recorded. Retrieval is achieved by selecting those components whose responses (with respect to the program) are closest to a pre-determined set of desired responses. This idea was originally called "behavioural sampling" by Podgurski and Pierce. A component is represented as a relation between programs and responses. This is because in general, a program execution can yield several responses (due to non-determinism) and a response may be evoked by more than one program. Formally, a component *C* can be declared as

C: program ←→response

A program p belongs to program is modeled as a sequence of calls on the component's interface. A response is a sequence of values in correspondence with a program. In effect, each program determines a context in which the behavior of a component is exhibited. The behavior of a component *C* is derived from the set of response sequences by removing those responses which are proper extensions of other responses. Whereas behavioral sampling technique did not necessarily collect all the possible execution responses but rather samples the responses over a number of executions, and exercised the most commonly used operations based on a probability distribution [7]. Thus, the behavior of a component *C* is the set of guaranteed responses to a program.

P: response →behavior

2.3. Behavioral matching by predicate logic

The description of software component based on facets. Among the facets which are considered for component description the important facet is behavior of the component. Generally, each source file consists of the comments as per the standards prescribed by the companies. The behavior of the component is extracted from the comments of the source code file and later these comments are converted to first order predicate logic i.e. describing a code in a formal method. An important point to be noted while considering the comments is that not all the comments are converted to first order predicate but the comments that includes information about the input ,output and some other important operations in the code are only converted. Since each line of code has a precondition and post condition it is not possible to consider all pre and post conditions, so we take into the consideration of only few pre and post conditions like input of the function, output of the function and some other important operations in the code of the function[6].

Example: Consider the component (binary search subroutine)

The binary search function works as follows: it takes an array or list as an input and a key value which is to be found as the output from the list. The prerequisite of the binary search is that the list should be in an sorted order. The list is further divided into two halves such that the merging of two lists gives the original list i.e., no loss of elements must happen. The function code searches the element in both the halves of the list so as to minimize the time.

PSEUDO CODE FOR BINARY SEARCH

```
//alist is the list of  integer elements
//item is the integer element to be found
//alist should be sorted form

def binarySearch(alist, item):
//assigning low and high indices
        first = 0
      last = len(alist)-1
      found = False
    // divides the list into two halves
      // searching in either of the divided arrays
      while first<=last and not found:

          midpoint = (first + last)//2
        if alist[midpoint] == item:
          found = True
        else:
          if item < alist[midpoint]:
            last = midpoint-1
          else:
            first = midpoint+1
              return found
```

//returns the position and item value found

Step 1:

The above source code file, we have the documentation i.e., comments regarding the code. We consider the comments related to the input , output and other important functions or operations performed. They are

- alist is the list of integer elements
- item is the integer element to be found
- alist should be sorted form
- divides the list into two halves
- returns the position and item value found

Step 2: The next step is to convert these English statements to first order predicate .

1. English sentence: alist is the list of integer elements
   First order predicate: is(alist, list)

2. English sentence: item is the integer element to be found
   First order predicate: is(item,integer)
3. English sentence: alist should be in sorted form

   First order predicate: sort(alist)

4. English sentence: divides the list into two halves
   First order predicate: equals((alist1.alist2),alist)
5. English sentence: returns the element is found or not
   First order predicate: is_in(item,alist1)

   is_in(item,alist2)

The behavior of the software component of binary search is described as follows:

- is(alist, list)
- is(item,integer)
- sort(alist)
- equals((alist1.alist2),alist)
- is_in(item,alist1)
- is_in(item,alist2)

3. PROPOSED WORK:

The preprocessing step for retrieval of a software component is the way of describing the software component. The dataset resulted from the real world code reuse repositories [9] plays a vital role for the algorithms used for retrieval. In general the procedure followed to retrieve a component will be the same as how the component is described .There are many traditional techniques say "keyword" search where a component is described with a set of keywords and the component is retrieved if we specify the keyword relevant to the component.

There are many techniques in literature which extract the behavior of components. Most of the existing techniques are based on formal mathematics which takes a specific model to extract the behavior. Model based techniques have to be different for different code reuse repositories and the drawbacks are that they are expensive as they follow some ML and have interoperability problem. The retrieval is appreciated only when the component retrieved can be substituted (reused) for other applications.

The proposed work is applied on diversified data repositories which extract the behavior from the documentation written in the source code .The comparision analysis and normalizing all code repositories is the major contribution of the proposed work. The dataset once normalized from different code repositories is achieved then the preprocessing step for retrieval is done.

1. Below is the Sample Customized source code where the comments have input and output along with some other important operations

```
//fn_arr is an array
// MAX_SIZE is an int
void insertion(int fn_arr[]) {
  int i, j, a, t;
  for (i = 1; i < MAX_SIZE; i++) {
    t = fn_arr[i];
    j = i - 1;
    while (j >= 0 && fn_arr[j] > t) {
      fn_arr[j + 1] = fn_arr[j];
      j = j - 1;
    }
    fn_arr[j + 1] = t;
    printf("\nIteration %d : ", i);
    for (a = 0; a < MAX_SIZE; a++) {
      printf("\t%d", fn_arr[a]);
    }
  }
  printf("\n\nSorted Data :");
  for (i = 0; i < MAX_SIZE; i++) {
    printf("\t%d", fn_arr[i]);
  }
}
//j is index
//swaps the j and j+1
//returns the sorted array
```

The comments are extracted and are converted to first order predicate and along with this behavior the other facets described are time(TC) and space complexity(SC), programming language (PL)and operating system(OS).

Behavior: is(fn_arr,array)#is(MAX_SIZE,int)#sort(fn_arr)#is(j,index)#swaps(j,j+1)

The description of the above component looks like

TC	SC	PL	OS	Behavior
O(n)	O(1)	c	Windows	is(fn_arr,array)#is(MAX_SIZE,int)#sort(fn_arr)#is(j,index)#swaps(j,j+1)

2. Consider the code repository of Java Standard Library(JSL) and a sample source code from JSL is shown below

```
/*
 * Copyright (c) 1994, 2008, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 */

package java.lang;
   /**
    * See the general contract of the <code>readFully</code>
    * method of <code>DataInput</code>.
    * <p>
    * Bytes
    * for this operation are read from the contained
    * input stream.
    *
    * @param      b     the buffer into which the data is read.
    * @param      off   the start offset of the data.
    * @param      len   the number of bytes to read.
    * @exception  EOFException  if this input stream reaches the end before
    *             reading all the bytes.
    * @exception  IOException   the stream has been closed and the contained
    *             input stream does not support reading after close, or
    *             another I/O error occurs.
    * @see        java.io.FilterInputStream#in
    */
```

The important comments considered here are starting with @param, @exception, @returns etc. and the remaining are deleted from code and this process is data reduction. The datatypes of @param are read from function header using Regular Expression. The above component is rewritten as

@param n the number of bytes to be skipped.

@return the actual number of bytes skipped.

@exception IOException if the contained input stream does not support

public final int skipBytes(int n) throws IOException

=>Is(n,int)

Skipped(bytes)

IOException(input stream)

The other facets described are time(TC) and space complexity(SC), programming language (PL)and operating system(OS)along with behavior and the description of the above component looks like

Table1: sample dataset from Java standard library

TC	**SC**	**PL**	**OS**	**Behavior**
O(1)	O(1)	Windows	Java	Is(n,int)# Skipped(bytes)# IOException (input stream)

In order to reuse software components the mining of code reuse repositories to some common form is required.This is known as discretization technique.The dataset retrieved from two different code reuse repositories mentioned above can be written as follows in a single dataset.

Table2: sample dataset from Customized dataset

TC	**SC**	**PL**	**OS**	**Behavior**
O(n)	O(1)	C	Windows	is(fn_arr,array)#is(MAX_SIZE,int)#sort(fn_arr)#is(j,index)#swaps(j,j+1)
O(1)	O(1)	Windows	Java	is(n,int)#Skipped(bytes)#IOException (input stream)

4. IMPLEMENTATION RESULTS

The results are extracted from two datasets (Table 1 and Table 2). One dataset is extracted from java standard library (JSL) and the other is customized dataset. A keyword search technique is applied on both datasets and the relevant retrieved components for "search" keyword are noted. The relevant components are nearly 85% on customized dataset and nearly 80% on standard java library.

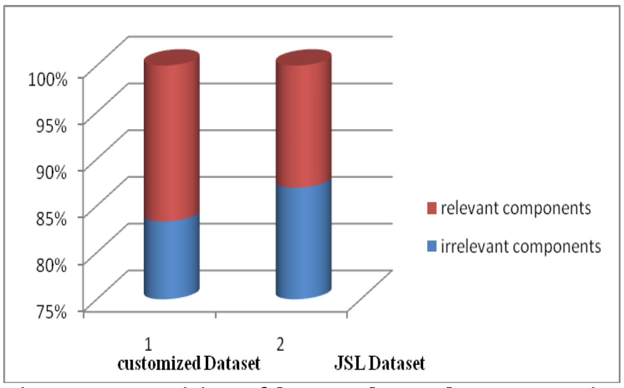

Fig. 1. Comparision of keyword search on customized dataset and JSL

The main purpose of the proposed work is to make the different datasets normalized .The above statistics shows that even in we integrate the different datasets from different code reuse repositories the results efficiency is not minimized.

5. CONCLUSION AND FUTURE SCOPE

There are many techniques in literature which extract the behavior of components. Most of the existing techniques are based on formal mathematics which can only be applied by the experts .There are even some informal techniques like predicate calculus which is explained in the Related work. The advantage of informal way of extracting the behavior of software components is the simplicity. The main focus of the proposed work is converting any real world dataset (diversified dataset) into behavior of the component (description of component) which further can be processed by the machine. Once the behavior of the component is extracted then any of the clustering or retrieval algorithms can be applied. This is the most important preprocessing step for the component clustering or retrieval. The future scope can be worked on converting the datasets into a different way rather than the behavior.

REFERENCES

[1] Swathy vodithala and Suresh pabboju,"Aspects related to the specifications of software components", International Journal of Computer Science & Engineering Technology (IJCSET), Vol. 5 , No. 03 Mar 2014.

[2] Book:BillCouncill ,"Definition of a Software Component and Its Elements", chapter 1.

[3] Book:Debayan Bose,"Component Based Development: Application In Software Engineering".

[4] Swathy vodithala,Niranjan Reddy and Preethi, "A Resolved Retrieval Technique for software Components", IJARCET, volume 1, issue 4, june 2012.

[5] Swathy Vodithala and Suresh Pabboju "A Keyword ontology for retrieval of software components", IJCTA ,10(19), 2017, pp. 177-182, International Science Press.

[6] Swathy Vodithala and Suresh Pabboju " A description of software reusable component based on behavior",ICTIS,2017 pp.602-609, Springer.

[7] Steven Atkinson, "Examining behavioral matching", Software Verification Research Centre.Department of Computer Science University of Queensland

[8] Amy moormann zaremski and Jeannette m. wing, "Specification matching of software Components", ACM Transactions on Software Engineering and Methodology, Vol. 6, No. 4, October 1997, Pages 333–369.

[9] Yunwen ye and Gerhard fischer,"Promoting reuse with active reuse repository systems",proceedings of 6th Internationl conference on software reuse(ICSR-6),Sprimger verlag,Austria,June 27-29,2000,pp:302-317