54

Rule-based Reasoning Approach for Reusable Design Component Retrieval

Sen-Tarng Lai[ab] and Chien-Chiao Yang[a]

[a]Dept. of Electronic Engineering, National Taiwan Institute of Technology, 43 Keelung Road, Section 4 Taipei, TAIWAN

[b]Telecommunication Laboratories, Ministry of Transportation and Communications, 9 Lane 74, Hsin-Yi Road, Sec. 4 Taipei, TAIWAN

Abstract

Each phase of software development life cycle has the potential for reuse. However, according to the viewpoint of cost and reusability, the design phase is the most suitable phase for packaging the reusable software components. In this paper, we describe the design component knowledge extraction and representation method for component retrieval. The design knowledge is based on the principle of structured design and object-oriented design. According to the valuable and clearly design component knowledge, we generate a rule-based system which contains the component matching rules and retrieval supporting rules. Using these rules not only can assist the potential reusers retrieve the more suitable design component but also can improve component retrieval performance.

Keyword Codes: D.2.m; H.3.3; I.2.4; I.2.m
Keywords: Software Reuse; Component Retrieval; Knowledge Representation; Rule-based System.

1. INTRODUCTION

The heart of large software project is design. Software design is conducted in two steps. Preliminary design is concerned with the transformation of requirements into data and software architecture. Detail design focuses on refinements to the architectural representation that lead to *detailed data structure* and *algorithmic representations* for software.

Code reuse is better understand and more prevalent by far than other software development phase of reuse[1]. Since code components have a high degree of specificity, the most highly reusable components tend to be small. A code component in a reuse library is likely to be of little value and the detailed design documents should be very valuable in understanding code component. Thus it is extremely important that detailed design documents associated with code modules to be a reusable design component [2-3]. This reusable design component is the most suitable software component for reuse.

In this paper, we detailed describe the important design knowledge for component classification and retrieval, and specify how to extract the knowledge from detailed design documents. We also propose the methods for represent the design component knowledge and transfer the design component knowledge into the rule-based system for assisting component's retrieval. In Section 2, we describe the existing software component's

M. Lee et al. (eds.), *Software Quality and Productivity*

representation approaches for classification and retrieval. In Section 3, we specify the important attributes of design component and package them into *a frame-based representation*. In Section 4, we explain the features of rule-based component retrieval system. At last, we make a summary and discuss our future works in Section 5.

2. LIMITATIONS OF EXISTING COMPONENT REPRESENTATION METHODS

In the Priteto-Diaz's faceted classification method which describe *the unit functional component classification* in coding phase [4-5], each component is characterized by a six-item structure consisting of

<function, object, medium, system type, functional area, setting>

and each tuple can only be specified by one term (keyword).

In [6], the authors applied the idea of conceptual dependency to represent software component descriptions and software component requests.

These methods have been published many years and applied to the many software library systems. There are also several drawbacks and limitations need to modify or enhance: (1) An action verb or a functional keyword just can represent the smaller components which process the primitive function. Larger numbers of smaller components forces the reuser to spend more time comparing components to determine the one best suited for the current application. (2) Internal data structure is a very important item for software component's retrieval, modification, and adaptation. They did not pay much attention to it. (3) Thesaurus is not a good approach for resolving the conceptual similar verbs in different applications. (4) They are not suitable for object-oriented design. For example, to represent the object of stack, it is necessary to separate three primitive functions in the faceted classification scheme of Priteto-Diaz and Freeman: <empty, character, table>, <push, character, table>, and <pop, character, table>. For resolving these limitations and drawbacks, we propose a flexibility knowledge representation method which describes in next Section.

3. COMPONENTS KNOWLEDGE REPRESENTATION

There are many attributes to represent the software design phase components. Only three important attributes are suitable for the criteria of software component retrieval. The three basic component's attributes are: *processing action*, *internal data structure* and *component's interface*. In this section, we will discuss how to represent, extract and package the component's attributes for component retrieval.

3.1. Action statements

In software detailed design phase, one of the task is describe the program logic. In component retrieval, it is meaningless for the reuser to describe the detailed logic of software components. They just concern whether the functional specifications of software components match their requirement or not. There are many approaches to represent the component function (such as keywords, natural language). It is difficulty to build the common recognition for using natural language to represent component function. Using an action verb or a functional keyword to represent the component function is too rough. An action statement can represent five or more program instruction statements. An action statement

contains a single strong action verb and a singular object (such as, open a file, delete a record). A series of action statements not only can represent the processing action of component but also can build the common recognition for component designers and component reusers.

The objective of thesaurus is to resolve the trouble of synonym. But the thesaurus did not consider the relationship of semantic and limitation of application domain. Based on the thesaurus, the component designers and the component users can not get common recognition. In this paper, we propose the uniform action verbs to represent the action statement of component processing action. Table 1 is the action verbs' table for action statements.

Table 1.
Action verbs' table for action statements

accept	delete	handle	put
add	dequeue	identify	queue
allocate	detach	increment	read
analyze	determine	initialize	record
build	display	insert	release
calculate	edit	issue	resolve
check	encode	locate	restore
clear	enqueue	link	scan
close	enter	load	schedule
complete	establish	look_up	search
construct	execute	merge	select
control	extract	modify	set
convert	find	move	store
copy	fix	obtain	transfer
create	format	open	translate
decrement	get	place	updat
			write

3.2. Internal data structure

Program consists of *data structure* and *algorithms* [7]. Thus it is not surprising that data and program structure are important and need to be properly related to achieve the goal of successful programming. Data structures not only affect the operation style of program algorithm but also contain information and are operated on during the execution of a program. Two software components doing exactly the same function may look entirely different, because they use different internal data structures. Clearly define data structure of software component can assist us to classify software component. In like manner, software component retrieval, modification and adaptation also need to use the attribute of internal data structure. It is important to specify the data structure of software components. The C++ Booch Components [8] provide classes for the domain independent data structure: *Bag, Deque, Graph, List, Map, Queue, Ring, Set, Stack, Variable_string* and *Tree*.

For enhancing forenamed data structure, we also consider the file structure (such as Sequential_File, Indexed_File, and Relative_File) and primitive data type (such as integer, character, real, array, record).

3.3. Component's interfaces

Each component can not stand alone. One of the software design process is establishing relationships and interconnections among components. In [9], the authors used I/O parameters to deduce the software component behavior and retrieve the retrieve software. Interfaces of software components are one of important attributes for retrieval.

3.4. Design component knowledge extraction

In [10], we use *module definition*, *module description*, and *module data representation* to specify the detailed design documents. These documents are very useful for module understanding, modification, adaptation. It is not necessary to utilize all of items of these documents as the attributes of component retrieval. We need extract the useful attributes which include internal data structure, action statements and component interface from these detailed design documents. The internal data structure can be gotten from module definition and module data representation. The action statements can be extracted from module logic description. The component interface can be abstracted from module definition and module data representation (see Figure 1).

Based on these valuable attributes, we can package the design component knowledge into *design knowledge frame*. In this frame-based knowledge representation, each slot can contain more than one item. According to the importance, each slot and item can be set different weight values for component retrieval (see Figure 2).

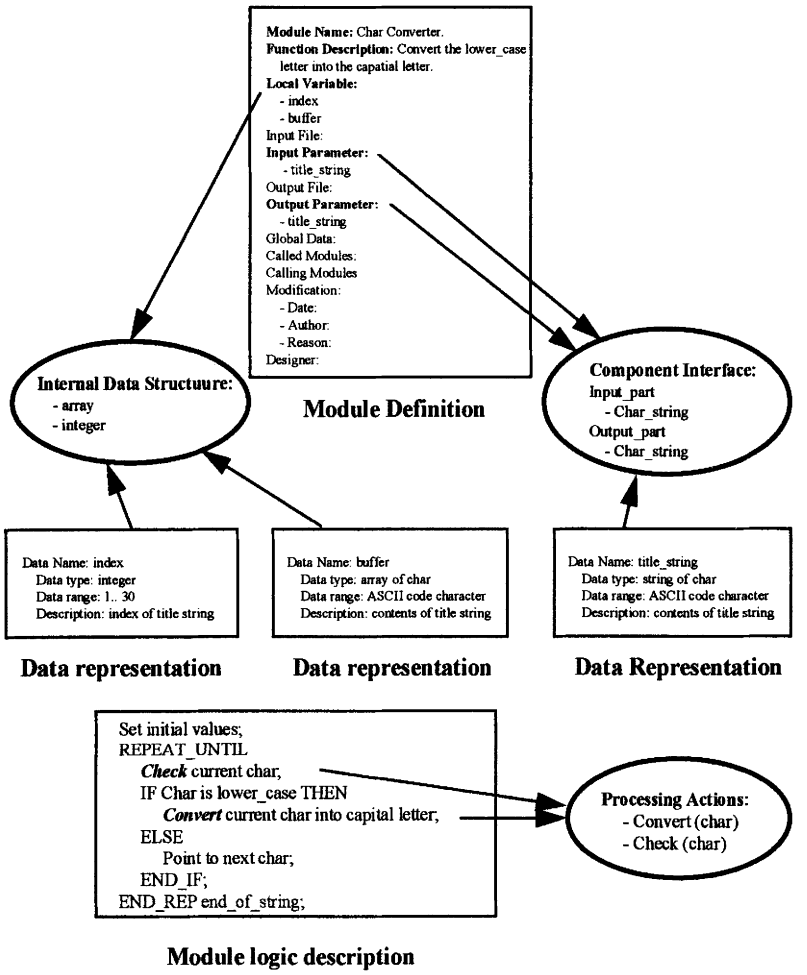

Figure 1. Extract the important attributes from detailed design documents

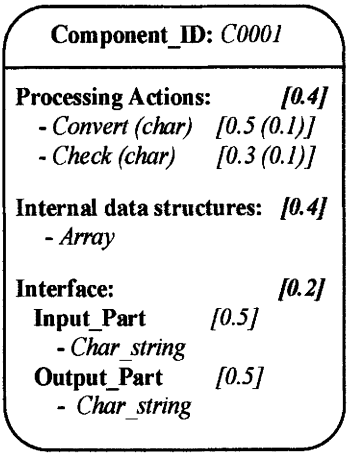

Figure 2. Frame-based representation for design component

4. COMPONENT RETRIEVAL RULES GENERATION

In the Section 3, we extract and package the design knowledge into a frame-based representation. In this Section, we will discuss the rule-based system generation which based on the frame-based representation. We give an example to describe the rule generation. This is a typical component which converts the lower-case letter into the capital letter (see Figure 2).

According to *frame-based knowledge* and *weight values*, system can generate the difference match ratio rules for component retrieval:

(a) The first set is fully match rules. If reusers provide the query information all meet the component attributes then these rules will be filed. For example,

Input (char) ^ Output (char) -> Interface (C0001) ***[0.5+0.5 = 1.0]***

Convert(char) ^ Check(char) -> Process(C0001) ***[(0.5+0.1) + (0.3+0.1) = 1.0]***

Data_Structure(array)^ Interface(C0001) ^ Process(C0001) -> Component (C0001)
[0.4 + 0.2 * 1.0 + 0.4 * 1.0 = 1.0]

(b) The second set is partial match rules. According to the query information of reusers, these rules can be filed and computed the match ratio for reference. For example,

Convert( ) ^ Check( ) -> Process(C0001) ***[(0.5 + 0.3) = 0.8]***

Data_Structure(array) ^ Process(C0001) -> Component (C0001) ***[0.4 + 0.4 * 0.8 = 0.72]***

Data_Structure(array) ^ Process(Convert) -> Component (C0001) ***[0.4 + 0.4 * 0.5 = 0.6]***

We also provide the more useful rules for assisting component retrieval before reasoning these basic rules. We call them as *retrieval supporting rules.* These rules are generated by analyzing and computing the appearance frequency of action statements in specific internal data structure of existing software components (see Table 2). Based on the retrieval supporting rules, reuser can be guided to modify the unreasonable query information and get the suitable software components. Reducing the unreasonable query information can great improve the performance of component retrieval.

Table 2.
The table of appearance frequency of action statements in specific internal data structure

Internal Data Structure	Action Statements	Frequence
Array		
	Search	97
	Get	53
	Check	47
	Convert	7
	:	:
	:	:
Bag		
:		
:		

5. CONCLUSIONS

To improve software quality and productivity are major benefits of software reuse. However, for increase efficiency, reuse should be considered at design time, not after the implementation has been completed. Design reuse is more valuable than code reuse. Design documents combine the knowledge and the experience from software design engineers. It is very important how to reuse the existing knowledge and experience. In this paper, we detailed describe the important design component knowledge for component retrieval, and specify how to extract the knowledge from detailed design documents. Finally, we propose the methods for represent the design component knowledge and transfer the design component knowledge into the rule-based system for assisting component's retrieval.

Four major advantages of rule-based component retrieval system are:

(1) *high hit-ratio* -- the uniform keywords can build the common recognition between component designers and reusers. According to the common recognition and extension of component knowledge items for retrieval, reusers have the high hit-ratio for component retrieval.

(2) *high adaptability* -- the design knowledge representation not only suit for structured design methodology but also can apply to the object-oriented design approach.

(3) *high maintainability* -- it is very easy to insert or delete a reusable software design component.

(4) *high performance* -- based on the history data and experience, this system can avoid the unreasonable query information and great improve the retrieval performance.

Some software metrics can help us extract and identify reusable software components. Including these metrics to the attributes of software component, can not only relies the characteristics of software components but also can help reuser select the suitable software components. Our future work will consider the metrics of software design. Based on the result of software design components' measurement, we can extract the more suitable components.

REFERENCES

1. LANERGAN, R.G. and GRASSO, C.A.: '**SOFTWARE ENGINEERING WITH REUSABLE DESIGNS AND CODE**', *IEEE Trans. Software Eng.*, 1984, Vol.10, (5), pp.498-501
2. JOHNSON, R. E. and FOOTE, B.: '**DESIGNING REUSABLE CLASSES**', *Journal of Object-Oriented Programming*, 1988, Vol. 1, (2), pp.22-35
3. TRACZ, W.: '**SOFTWARE REUSE MYTHS**', *ACM SIGSOFT Software Engineering Notes*, 1988, Vol. 13, (1), pp.17-21
4. PRIETO-DIAZ, R. and FREEMAN, P.: '**CLASSIFYING SOFTWARE FOR REUSABILITY**', *IEEE Software*, 1987, Vol.4, (1), pp. 6-16
5. PRIETO-DIAZ, R.: '**IMPLEMENTATION FACETED CLASSIFICATION FOR SOFTWARE REUSE**', *Proc. of the 12th Internalational Conference on Software Engineering*, March, 1990, pp. 300-304
6. WOOD, M. and Sommerville, I.: '**AN INFORMATION RETRIEVAL SYSTEM FOR SOFTWARE COMPONENTS**', *Software engineering Journal*, 1988, Vol.3, (5), pp. 198-207
7. WIRTH, N.: '**ALGORITHM + DATA STRUCTURES = PROGRAM**', (Prentice-Hall, 1976)
8. BOOCH, G.: '**OBJECT-ORIENTED ANALYSIS AND DESIGN**', (Benjamin/Cummings, 1994)
9. PODGURSKI, A. and PIERCE, L.: '**RETRIEVING REUSABLE SOFTWARE BY SAMPLING BEHAVIOR**', *ACM Trans. on Software Engineering and Methodology*, 1993, Vol.2, (3), pp. 286-303
10. LAI, S.T. and CHOU, L.Y.: '**MODULE DESIGN DOCUMENT ASSISTANT SYSTEM**', *TL Technical Journal*, 1987, Vol.17, (2), pp. 93-108 (in Chinese)