A Formal Approach for Specification and Classification of Software Components

Sathit Nakkrasae [1,2]

[1] Center for Advanced Computer Studies (CACS), The University of Louisiana at Lafayette, Lafayette, LA 70504, U.S.A.

khunsathit@hotmail.com

Peraphon Sophatsathit [2]

[2] Advanced Virtual and Intelligent Computing (AVIC) Center, Faculty of Science, Chulalongkorn University, Bangkok, 10330,Thailand

Peraphon.S@Chula.ac.th

ABSTRACT
Software components have played an important role in modern software and system development. The main contribution of software component is reuse which, in essence, helps reduce development cost and time and increase productivity at the expense of usage problems. Typical component reuse problems are found in large software component repositories due to inefficient component look up and difficulty in furnishing a comprehensive description for component identification, specification, and classification. This paper presents a formal approach based on the well-established object-oriented paradigm to resolve the above difficulty. As each component is searched, an additional precaution is incorporated to ensure the correct result being retrieved. This extra step is known as software certification.

General Terms
Measurement, Design, Standardization, Theory.

Keywords
Reuse, component identification, specification, classification, software certification.

1. INTRODUCTION
In a large component repository, software component retrieval is accomplished through some classification schemes. Users supply as much relevant descriptions as possible for closest match of the desired component. The underlying implementation details are often transparent to the users. As such, it is imperative for every archived component that it be correctly identified, classified, and stored for subsequent retrieval.

The fundamentals of software components primarily rest on object-oriented concepts, but aim at much larger specification than that of a single object. Although the concepts encompass various reusability provisions, component classification and archival principles are still under investigation. We propose a well-defined component classification framework based on conventional object-oriented paradigm and formal approach for systematic component storage and retrieval.

The remaining of this paper is organized as follows. Section 2 discusses related papers on identification, specification, and classification of software components. Other related topics are also incorporated. Section 3 establishes our research methodology for identification and specification of software component. The application of the proposed approach in component classification based on structural, functional and behavioral properties is presented in Section 4. Our final thoughts and future work are discussed in Section 5 and Section 6, respectively.

2. RELATED PAPERS
Software component library construction, as presented in [14] using information retrieval techniques to automatically assemble various components, can be divided into two steps. First, attributes are automatically extracted from natural language documentation by means of an indexing scheme. Second, a hierarchical indexing scheme is generated using a clustering technique based on analysis of natural language documentation obtained from manual pages or comments. Despite being a rich source of conceptual information, natural language is not a rigorous language for specifying the behavior of software components. A formal specification language can thus be utilized to serve as a precise contract and a means for communication among software clients, specifiers, and implementers [10].

The MAPS system [17] applies formal specifications termed case-like expressions to specify software modules. MAPS exploits the unification capability to search through reusable modules in the library. Jeng, et. al, [11] utilizes formal methods to specify software component in a hierarchically organized library. However, both approaches still have granularity limitations.

Hong and Kim [9] propose three classification methods to systematically organize the components according to their formal specification so developed, namely, the enumerative method, the facet method, and the information retrieval method using clustering technique. The first and the last methods have some drawbacks as mentioned in [9]. Thus, we adopted the second method with some adaptation to be used as a basis for our specification derivation. Some of the related works [1,2,5,6,21,23] also investigate on such issues as component reuse, formal method for component reuse, and classification of software components.

3. PROPOSED METHODOLOGY

The process of software component identification and specification is based on three aspects, namely, structural, functional, and behavioral properties. We will employ component modeling technique (CMT) to specify and construct a visual component model using the Unified Modeling Language (UML) [15]. The essence of this approach is to capture as much information pertaining to the fundamental properties of the component as possible. Such information is then described through formal specification by means of Z language [3,27]. Once the components are identified and specified, classification process is carried out in two stages: coarse and fine grains, according to the above properties. The degree of classification accuracy is measured via existing component reusability metrics [18]. The final analysis of classified components is validated through a well-defined certification process [4,20,24,25,26] based on their reuse quality and usefulness.

3.1 Component Identification

The basic premise of software components rests on the notions of reuse building blocks. Every component is made up of zero or more subcomponent in the form of concrete classes. Thus, a component can be regarded as a self-contained complex entity consisting of a subcomponent, a class, or a family of subcomponents or classes, common data, and common methods. Each part is linked to its structurally related subcomponents or classes defined by some interaction, which eventually connected to the outside world via the interface. Thus, any component can be linked to other components via different interaction. The behavioral interaction among these subcomponents or classes is described in the form of transactions. This procedure is recursively applied to create and identify component interrelationships.

The following definitions from [7,12,13,15,19,22] will be used in subsequent library analysis and composition.

3.2 Component Modeling Technique (CMT)

Component Modeling Technique adheres to the “three views of a system” paradigm, namely, structure (or static model), behavior (or dynamic model), and function (or functional model). Each model is used to build a set of component views defined on a domain. As stated in 3.1, it is possible to recursively represent the embedded component view based on the covering domain. We will discuss the configuration of each model to establish our formal approach and to denote the component specification in the sections that follow.

3.2.1 Structural Model

The CMT Structural model consists of:

1) Component box with small rectangles representing methods on one side and little lollipops connected by solid line representing the interface on the other side,
2) Internal box representing classes with a mandatory class name, optional attribute name or declarations, and optional operation names and declarations,
3) Class relationships,
   a) Dashed line representing dependency relationship,
   b) Solid line representing association relationship,
   c) Solid line with opened arrow representing generalization relationship, and
   d) Dashed line with closed arrow representing realization relationship.
4) Component relationships,
   a) Solid line with closed arrow representing uses relationship,
   b) Dashed line with opened diamond representing implements relationship, and
   c) Dashed line with closed diamond representing extends relationship.
5) Bracketed textual items denoting constrains.

3.2.2 Functional Model

The Functional model in CMT specifies how operations derive output values from input values without regard to the order of computations. Components process the inputs according to their operational specification to yield the designated outputs. In this paper, we will employ UML diagrams for our discussion.

3.2.3 Behavioral Model

The Behavior model refines the activity model, proposed capabilities, and component capability requirements. Some capabilities may be fulfilled by a combination of multiple behaviors. The behavior model is divided into two parts as follows:

a) Component-Interaction part that shows the messages (behavioral name and/or message argument) sent between components (or subcomponents or classes).
b) State Transition part that presents the state transitions of each component (or subcomponent or class) or the interaction between the components (or subcomponent or class).

A state can be viewed as an equivalent class or subcomponent of attribute values and association values of an object class or subcomponent. This equivalent class or subcomponent is formed under a relation of “same behavior” with respect to the real world situation or requirements being modeled. The state thus defines a “transitory subtype” of the object class or subcomponent to which it belongs.

State description can be incorporated in the graph, based on David Harel, et. al [12], that consists of a name, optional internal actions (instantaneous operations), activities (operations that take time to complete), optional entry (trigger) and exit actions, and edges representing transitions between states.

Figure 1 shows the model of a component Array_Stack_Data_ Structure consisting of two subcomponents (Array and Stack), common classes (Method class and Attribute class), and their interactions.

3.3 Component Specification

In order to establish formal component specification, the syntax and semantics must be analyzed and represented. This section presents mathematical representations by means of Z specification language to capture the three views of CMT in formal component specification.

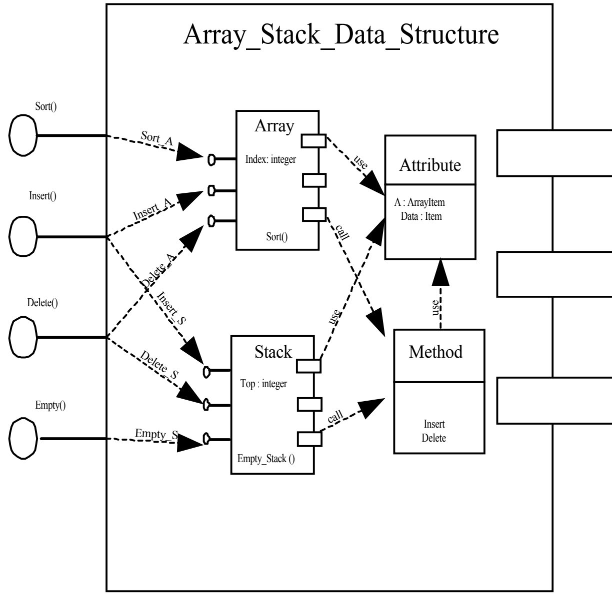

Figure 1. Component Model

A software component (abbreviated SC) is a 5-tuple element having the form:

SC := (*n*, *SSC, CL, SIG , I*) where
n : is a unique name of the software component
SSC : (sub-software component) is a set of software component
CL = *MC* ∪ *CC* where
MC : member class (set of class)
CC : common class (set of class)
CC = *Met* ∪ *Atb where*
Met : method class (set of class)
Atb : attribute class (set of class)
SIG : is a set of 6-tuple signature describing the operation (function) having
SIG := { (*nf, In, Local, Out, Pre, Post*) } where
nf : name of operation (function)
In : list of input parameter
Local : list of local variable
Out : list of output parameter
Pre : Expression
Post : Expression
I : is a set of 3-tuple interaction describing the transaction such that
I := {(*IS* ,*ID* , *Bh*)} where
IS : Interaction source
ID : Interaction destination
Bh : is a set of 3-tuple behavioral part describing the state and action between *IS* and *ID*) having
Bh : = { (*nb*, *St* , *Ac*) }
Where
nb : name of behavior
St : set of state
Ac : set of action

Based on the above definitions, we developed standard Z notations for software component as follows:

Step 1: Denoting basic type sets which are declared and enclosed in square brackets as follows:
[Operational name]
[Input parameter]
[Output parameter]
[Local variable]
[Expression]
[Behavioral name]
[Component name]
[Class name]
[Request]
[Action]
[State]
[Class]
[string]

Step2: Denoting composite/aggregate type sets as follows:
Member class : F classes
Method class : F classes
Attribute class : F classes
Common class: method class ∪ attribute class
Interaction source : component name ∪ class name ∪ operational name
Interaction destination : component name ∪ class name ∪ operational name
ProperComponent_name : set of Component name
ProperOperational_name : set of Operational name
ProperBehavior_name : set of Behavioral name

Step3: Denoting software component, λ structure as follows:
λ == { *SC* : (*n*, *SSC* , *CL*, *SIG* , *I*) |
∀ *n* : Component name
SSC : P (set of Software component)
CL : P Class
SIG : P *Op*
I : P *Tr* •
(n, SSC , CL, S , I) ∈ SC =>
n ∈ ProperComponent_name ∧
SSC ∈ F (set of software component) ∧
CL ∈ Common class ∪ Member class ∧
SIG ∈ F *Op* ∧ *I* ∈ F *Tr* }

Step4: Denoting component signature, σ as follows:
σ =={*Op* : set of (*nf, In, Local, Out, Pre, Post*) |
∀ *nf* : Operation name
In : P Input parameter
Local : P local variable
Out : P Output parameter
Pre : Expression
Post : Expression •
(nf, In, Local, Out, Pre, Post) ∈ Op =>
nf ∈ ProperOperational_name ∧
In ∈ F Input parameter ∧
Local ∈ F local variable ∧
Out ∈ F output parameter ∧
Pre ∈ expression ∧
Post ∈ expression }

Step5: Denoting functional interaction, ι as follows:

ι == { *Tr* : set of (*IS, ID, Bh*) |
∀ *IS* : Interaction source
ID : Interaction destination
Bh : P *B* •
(IS, ID, Bh) ∈ Tr =>
IS ∈ Interaction source ∧
ID ∈ Interaction destination ∧ *Bh* ∈ F *B* }

Step6: Denoting component behavior, β as follows:

β == { *B* : set of (*nb*, *St* , *Ac)* |
∀ *nb* : Behavior name
St : P State
Ac : P Action •
(nb, St, Ac) ∈ B =>
nb ∈ ProperBehavior_name ∧
St ∈ F state ∧ *Ac* ∈ F action }

Step7: Combining all notations (step3 – step6) into Z schema as shown in Figure 2-5

SpecifySoftwareComponent

n? : Component name
SSC? : P (set of software component)
CL? : P Class
SIG? : P *Op*
I? : P *Tr*
SC! : set of λ

---

n? ∈ F string ∧ *SSC?* ∈ F (set of software component) ∧
CL? ∈ Common class ∪ Member class ∧
SIG? ∈ F *Op* ∧ *I?*∈ F *Tr* ∧ *SC!* ∈ set of λ

Figure 2. Component structural specification

SpecifySignature

nf? : Operation name
In? : P input parameter
Local? : P local variable
Out? : P output parameter
Pre? : expression
Post? : expression
Op ! : σ

---

nf? ∈ F string ∧
In? ∈ F input parameter ∧
Local ? ∈ F local variable ∧
Out ? ∈ F output parameter ∧
Pre ? ∈ expression ∧
Post ? ∈ expression ∧ *Op!* ∈ σ

Figure 3. Signature specification

SpecifyInteraction

IS? : Interaction source
ID? : Interaction destination
Bh? : P *B*
Tr! : ι

IS? ∈ F string ∧
ID? ∈ F string ∧
Bh? ∈ F *B* ∧
Tr! ∈ ι

Figure 4. Functional specification

SpecifyBehavior

nb? : behavior name
St? : P state
Ac? : P action
B! : β

---

nb? ∈ F string ∧ *St ?* ∈ F state ∧
Ac? ∈ F action ∧ *B!* ∈ β

Figure 5. Behavioral specification

Other definitions such as constants and expressions can be included in the declaration section and the predicate section, respectively. Figure 6-9 demonstrate a step by step component specification derivation of the Array_Stack_Data_Structure example given in Figure 1.

Specify Static_Sequential_Data_Structure

Array_Stack_Data_Structure? : Component name
{Array?, Stack?} : P (set of software component)
{Method?, Attribute?} : P Class
{Sort()?, Insert()?, Delete()?, Empty()?} : P *Op*
{Sort_A?, Insert_A?, Delete_A?…} : P *Tr*
Array_Stack_Data_Structure Component! : set of λ

Array_Stack_Data_Structure? ∈ F string ∧
{Array?, Stack?} ∈ F (set of software component) ∧
{Method?, Attribute?} ∈ Common class ∪ Member class ∧
{Sort()?, Insert()?, Delete()?, Empty()?} ∈ F *Op* ∧
{Sort_A?, Insert_A?, Delete_A?} ∈ F *Tr* ∧
Array_Stack_Data_Structure Component! ∈ set of λ

Figure 6. Example of Structural specification

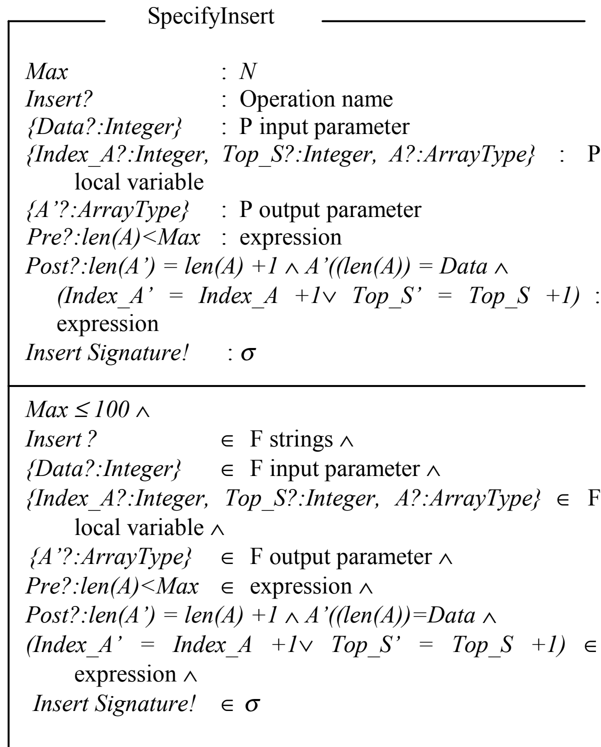

Figure 7. Example of Signature specification

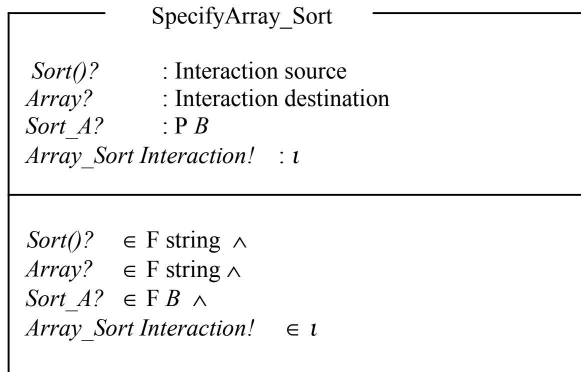

Figure 8. Example of Functional specification

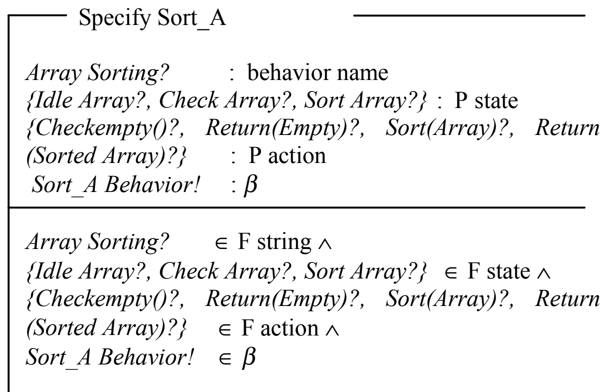

Figure 9. Example of Behavioral specification

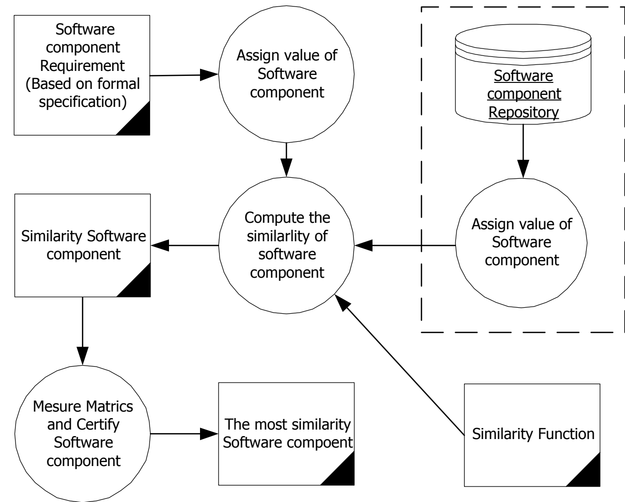

Figure 10. Software reuse model

4. COMPONENT CLASSIFICATION

We will employ the standard software component notations identified and represented in the previous section as a basis for component classification. We will first examine how software component is reused through the reuse model in order to establish a classification framework over the component applicable domains. A formal classification approach will then be presented, along with a simple example to demonstrate the applicability of the proposed framework.

4.1 Software Reuse Model

The software reuse model encompasses a repository which stores formal specifications of software components and retrieval mechanisms to facilitate component check-in/check-out during the development process. The underlying principle of the proposed classification scheme relies on component similarity comparison that is derived from a user-defined classification function. This offers a quantitative technique to enumerate the component suitability. Assessment begins by evaluating the requirements specification of the designated component and the components stored in the repository to arrive at a similarity value. The process is repeated over the entire classes of component search domain. If there exists more than one matching component, a certification step will be applied to select the closest similar candidate. This process is depicted in Figure 10.

4.2 Classifying Software Components

The three properties used to classify software components are component structure, function, and behavior. The component structure is made up of component name, subcomponent name, class name, signature, and interaction name. The component function consists of function name, input parameter, local variable, output parameter, and pre/post expressions. The component behavior is constructed from behavior name, state name, and action name. These properties are used to evaluate the desired component by applying a user-defined assessment function on each component specification. The result is a similarity value. The most straightforward assessment/evaluation is obtained from component structure, followed by component function, and component behavior, respectively.

The following notations are defined for use in component classification assessment. Let

- WS_s , WS_f , WS_b be the weight of structure, function, and behavior similarity, respectively, and $0 \leq WS \leq 1$.
- $Ø_s$, $Ø_f$, $Ø_b$ be the degree of significance of structure, function, and behavior, respectively, depending on the underlying system environment. The summation of $Ø_s$, $Ø_f$,and $Ø_b$ is equal 1. In this paper, we don't care about $Ø_s$, $Ø_f$, and $Ø_b$. For easily calculate, we will let : $Ø_s = Ø_f = Ø_b = 1$.

Given two components

$Cn = (C_{1(s)}, C_{2(f)}, C_{3(b)}, \ldots, C_{n(\ldots)})$

and

$Cn' = (C'_{1(s)}, C'_{2(f)}, C'_{3(b)}, \ldots, C'_{n(\ldots)})$

where

n is number of properties that we use to compute the similarity. In this paper, we define n = 3. If additional properties are necessary, such as non-functional, quantitative and qualitative properties, etc., n > 3,

$C_{1(s)}$ and $C'_{1(s)}$ denote the structure property of component *Cn* and *Cn'*, respectively,

$C_{2(f)}$ and $C'_{2(f)}$ denote the functional property of component *Cn* and *Cn'*, respectively,

$C_{3(b)}$ and $C'_{3(b)}$ denote the behavioral property of component *Cn* and *Cn'*, respectively, and

$C_{n(\ldots)}$ and $C'_{n(\ldots)}$ denote the nth property of component *Cn* and *Cn'* respectively.

For brevity, we will replace ***Cn*** by ***C***, representing the three properties of individual component.

Definition 1: (Similarity)

$Sim_{simdet}(para1, para2) \rightarrow [0,1]$ where *Sim* is the similarity function that returns the values between 0 and 1, having 0 denoted no similarity and 1 denoted exact match.

$_{simdet}$ is the similarity assessment of software component ($_c$), structure ($_s$), function ($_f$), or behavior ($_b$)

para1 and *para2* are attributes established in Section 3.3.

Each parameter is associated with an equivalent cluster (eq_cluster). For example, if *para1* denotes a stack and the stack is an equivalent cluster of LIFO, then eq_cluster (*para1*) = LIFO. A set of equivalent clusters can be predefined within the system. The number elements in the set of equivalent clusters for a given component repository must be finite.

$Sim_{simdet}(para1, para2)$ = 1 if para1 and *para2* are in the same eq_cluster, or

$Sim_{simdet}(para1, para2)$ = 0 if para1 and *para2* are in different eq_cluster.

Definition 2: (Software Component Similarity)

$$Sim_c(C, C') = Ø_s WS_s + Ø_f WS_f + Ø_b WS_b = \left[\sum_{i = s, f, b} Ø_i WS_i \right] / n$$

Let *AS* be structural specification of *C*, *Q* be query requirements specification of *C'*, and n_s be the number of structure members.

Definition 3: (Structural Similarity)

$$Ø_s WS_s = Ø_s (Sim_s(AS, Q)) = Ø_s \left[(Sim_s(AS_n, Q_n)) + (Sim_s(AS_{SSC}, Q_{SSC})) + (Sim_s(AS_{CL}, Q_{CL})) + (Sim_s(AS_{SIG}, Q_{SIG})) + (Sim_s(AS_I, Q_I)) \right] / n_s$$

Let *AF* be functional specification of *C* , *Q* be query requirements specification of *C'*, n_f be the number of function members, and *m* be the number of functions in component.

Definition 4: (Functional Similarity)

$$(Sim_f(AF, Q)) = \left[(Sim_f(AF_{nf}, Q_{nf})) + (Sim_f(AF_{in}, Q_{in})) + (Sim_f(AF_{local}, Q_{local})) + (Sim_f(AF_{out}, Q_{out})) + (Sim_f(AF_{pre}, Q_{pre})) + (Sim_f(AF_{post}, Q_{post})) \right] / n_f$$

$$Ø_f WS_f = Ø_f \left[\sum_{j=1..m} (Sim_{fj}(AF, Q)) \right] / m$$

Let *AB* be behavioral specification of *C*, *Q* be query requirements specification of *C'*, n_b be the number of behavioral members, and *p* be number of behavior in component.

Definition 5. (Behavioral similarity)

$$(Sim_b(AB, Q)) = \left[(Sim_b(AB_{nb}, Q_{nb})) + (Sim_b(AB_{St}, Q_{St})) + (Sim_b(AB_{Ac}, Q_{Ac})) \right] / n_b$$

$$Ø_b WS_b = Ø_b \left[\sum_{k=1..p} (Sim_{bk}(AB, Q)) \right] / p$$

The classification process may yield more than one classification. In which case, component certification is required based on suggested procedures given by [4,16,18,20,24,25,26]. Thus, the most similar of software component can be selected.

4.3 An Example

We will demonstrate graphical views of the proposed approach through an example. Figure 11 depicts three software components, namely, C'1, C'2, and C'3, residing in the repository. The requirements of target component (C) are to be computed. Figure 11a, 11b, and 11c depict the structural, functional, and behavioral similarity of.C'1, C'2, C'3, and C, respectively. The circles and corresponding squares represent the method and the behavior (interaction) of the component. Figure 11d shows the component similarity. The methods and interactions of the similar components are depicted in gray and black colors.

The values of component similarity can be computed as follows:

Structure similarity = no. of members in the same eq_cluster / no. of members

= 5 / 5 = 1

Function similarity = (Σ similarity values of the method) / target required value

= (4/6 + 5/6 + 3/6) / 7

= 0.357

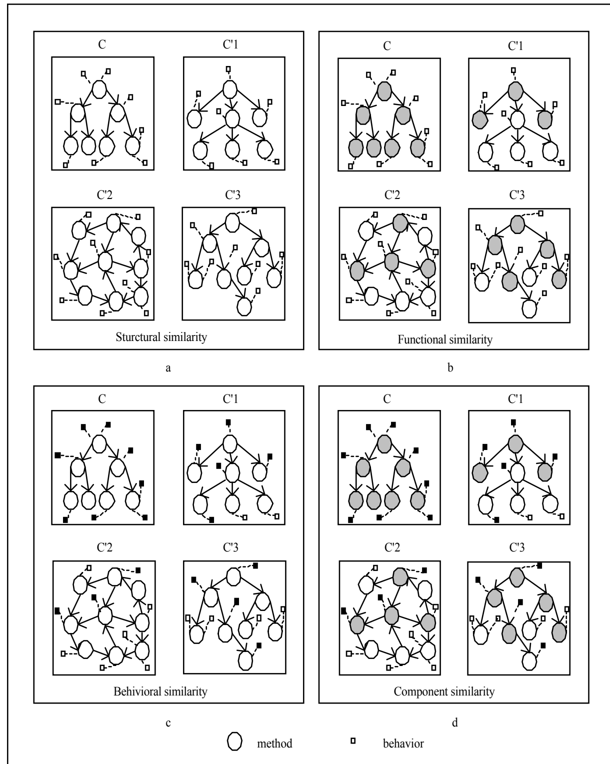

Figure 11. Example of Component Similarity

Behavior similarity = (Σ similarity values of the behavior) / target required value

= (2/3 + 2/3 + 2/3 + 3/3 + 2/3) / 8

= 0.459

The sum of similarity = (1+ 0.357 + 0.459) / 3

= 0.605.

The results are shown in Table 1.

Table 1. Similarity Comparison

	Structure	Function	Behavior	Sum
C'1	1.0	0.357	0.459	0.605
C'2	1.0	0.429	0.333	0.587
C'3	1.0	0.571	0.458	0.676

5. CONCLUSION

We have demonstrated the viability of formal approach using Z specification and CMT guidelines with the help of a simple array/stack example to define and classify existing software in to components based on their structural, functional, and behavioral properties. Simple as it may appear, the similarity classification was computed using information obtained from the equivalent clusters arranged manually. The assessment so obtained is further classified according to existing procedures to ensure the closest similarity of the component being retrieved. It is essential that the rigor and correctness of formal approach to software component identification, specification, and classification can ease the burden of software reuse in today's cyber-pace development life cycle. As a consequence, the ever-growing software development cost and time can be reduced, as well as considerable improvement on the end-product quality.

6. FUTURE WORK

In addition to specification and classification processes proposed in this paper, we are investigating suitable metrics, clustering techniques such as neural networks and fuzzy logic [8], and certification guidelines that will help validate the proposed classification process. It is hoped that software component research endeavors will extend to incorporate formal approach as a verification tool to properly group and certify software components. As such, the future of COTS development will be more widely practiced and accepted in the same manner as its hardware counterparts.

7. ACKNOWLEDGMENTS

We would like to express our appreciate to Assoc. Prof. Dr. William R. Edwards, Jr. of CACS at UL and Mrs. Wiphada Wettayaprasit at AVIC for their insight discussions, constructive suggestions, and correction of our paper.

8. REFERENCES

[1] 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, 333-369.

[2] L. Baum, and M. Becker. Generic components to foster reuse. In Proceedings of the 37th International Conference on Technology of Object-Oriented Languages and Systems, TOOLS-Pacific 2000, 266 –277.

[3] Ben Potter, Jane Sinclair, and David Till. An Introduction to Formal Specification and Z, 2nd Edition. Prentice Hall, England, 1996.

[4] G. Caldiera, and V. R. Basili. Identifying and qualifying reusable software components. Computer, Volume: 24 Issue:2, Feb. 1991, 61 –70.

[5] Chao-Tsun Chang, W.C. Chu, Chung-Shyan Liu, and Hongji Yang. A formal approach to software components classification and retrieval. In Proceedings of the 21st Annual International Computer Software and Applications Conference, 1997, COMPSAC '97, 264 –269.

[6] P. Chen, R. Hennicker, and M. Jarke. On the retrieval of reusable software components. In Proceedings of the 2nd International Workshop on Advances in Software Reuse Software Reusability, 1993, 99 –108.

[7] Clemens Szyperski. Component Software: Beyond Object-Oriented Programming. Addison-Wesley, England. 1998.

[8] S.A. Ehikioya. A formal model for the reuse of software specifications. In IEEE Canadian Conference on Electrical and Computer Engineering, 1999, Volume: 1, 283-288.

[9] S.B.Hong, and Kapsu Kim. Classifying and retrieving software components based on profiles. In Proceedings of International Conference on Information, Communications and Signal Processing (ICICS), 1997, Volume: 3, 1756 – 1760.

[10] M. Jeannette Wing. A specifier's Introduction to Formal Methods. IEEE Computers, September 1990, pp. 8-24.

[11] Jun-Jang Jeng, and B.H.C. Cheng. Using Formal Methods to Construct a Software Component Library. In Lecture Notes in Computer Science, vol. 717. In Proceedings of the 4th European Software Engineering Conference, September 1993, 397-417.

[12] Kevin Lano, and Howard Haughton. Object-Oriented Specification Case Studie. Printice Hall, England.

[13] Klaus Bergner, Andreas Rausch, and Marc Sihling. Componentware-The big picture. http://www.sei.cmu.edu/cbs/icse98/papers/p6.html, present at the 1998 International Workshop on Component-Based Software Engineering.

[14] Y. S. Maarek, D. M. Berry, and G.E. Kaiser. An Information Retrieval Approach for Automatic Constructing Software Libraries. IEEE Transactions on Software Engineering, August 1991, 800-813.

[15] Meilir page-jones. Fundamentals of Object-Oriented Design in Uml. Addison-Wesley, United states, 2000.

[16] J. Morris, G. Lee, K. Parker, G.A. Bundell, and Chiou Peng Lam. Software component certification. Computer, Volume: 34 Issue: 9, Sept. 2001, 30 –36.

[17] F. Nishida, S. Takamatsu, Y. Fujita, and T. Tani. Semi-Automatic Program Construction from Specification Using Library Modules. IEEE Transactions on Software Engineering, 1991, 853-870.

[18] Jeffrey S. Poulin. Measuring software reusability. In Proceedings of the 3rd International Conference on Software Reuse: Advances in Software Reusability, 1994, 126 –138.

[19] Roger S. Pressman. Software Engineering, A Practitioner's Approach, 4th Editon. The McGraw-Hill Companies, Inc., United States,1997.

[20] S.L. Rohde, K.A. Dyson, P.T. Geriner, and D.A. Cerino. Certification of reusable software components: summary of work in progress. In Proceedings of the 2nd IEEE International Conference on Engineering of Complex Computer Systems, 1996, 120-123.

[21] M. Saeki. Behavioral specification of GOF design patterns with LOTOS. In Proceedings of the 7th Asia-Pacific Software Engineering Conference, APSEC 2000, 408-415.

[22] Stephen H. Edwards, David S. Gibson, Bruce W. Weide, and Sergey Zhupanov. Software Component Relationships. http://www.umcs.maine.edu/~ftp/wisr/wisr8/papers/edwards/edwards.html.

[23] I.Tvrdy. Formal approach to reusable formal specifications. In Proceedings of the 6th Mediterranean Electrotechnical Conference, 1991, 1045 –1048.

[24] J.M. Voas. Certifying off-the-shelf software components. Computer, Volume: 31 Issue: 6, June 1998, 53 –59.

[25] C.Wohlin, and B. Regnell. Reliability certification of software components. In Proceedings of the 5th International Conference on Software Reuse, 1998, 56 – 65.

[26] C. Wohlin, and P. Runeson. Certification of software components. IEEE Transactions on Software Engineering, Volume: 20 Issue: 6, June 1994, 494 –499.

[27] J. C. P. Woodcock. Structuring specifications in Z. Software Engineering Journal, Volume: 4 Issue: 1, Jan. 1989, 51–66.