CONTENT-BASED FRAMEWORK FOR COMPONENT LIBRARIES

Franz Kurfess
Ching Kang Cheng
Department of Computer Science, California Polytechnic State University, San Luis Obispo, CA 93407, U.S.A.

Z.Y. Wang
Tumbleweed Communication Corp., Redwood City, CA 94063, U.S.A.

L. Jololian
New Jersey City University, NJ 07305, U.S.A.

The construction of large software systems from existing components requires efficient access to possibly large collections of such components. In this paper, we present a content-based framework to organize this large collection of components in a semi-automatic fashion, according to an extensible user-defined ontology. Neural associative memories are used for fast, similarity-based access. Relevant characteristics of components are extracted and stored as a ``signature'' in such an associative memory, and during retrieval the closest matches to the query are indicated in very short time. In addition to content-based characteristics such as keywords, function and variable names, comments, and the location of the component in the class hierarchy, this method can be easily expanded to include usage characteristics, resource requirements, or other task- or customer-specific criteria.

Keywords: *components, neural associative memories, ontology.*

1. Component-Based Systems

The demand for new specialized software systems is on the increase, with the size and complexity of these systems challenging the effectiveness of the traditional software engineering development methodologies. To meet this challenge, organizations must be increasingly agile, and quickly respond to market conditions. Component technology is an effective methodology that provides powerful means of building systems quickly from a library of existing components. As libraries grow larger, the organization and retrieval task becomes more cumbersome.

In this paper we propose a framework to identify components according to syntactic, semantic, and pragmatic criteria. This framework consists of three main modules: a signature extraction engine (SEE), a facet-based component representation scheme (FCRS) and a neural associative memory (NAM). SEE constructs meaningful signatures according to the ontology from the documentation affiliated with the components. The signatures retrieved from components are categorized to the respective facets, defined in FCRS. NAM provides instant, similarity-based access according to the

criteria provided via the facet representation, resulting in a very fast and reasonably powerful retrieval mechanism.

1.1. Component Definition

Among the variety of definitions for components proposed in the literature, we will use here the following one from (Szyperski, 1997):

> *A component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third party composition.*

In this definition, code segments are explicitly included as "software components"; it is possible, however, to also include components that are not necessarily executable software units, such as templates, design patterns, system frameworks, or other units relevant for the design and construction of system. This definition is more comprehensive than others in the sense that it includes different types of reusable units within the scope of components. Components can be seen on two levels, the design level and the implementation level. At the design level, components are used to describe the functionality of a desired software system and how the system can be built based on available components. The types of components found at this level are system architectures, frameworks, templates and design patterns. At the implementation level, components are encapsulated code segments, which are self-contained and perform specific functions. This kind of components is also referred to as "commercial off-the-shelf components," or COTS components, for short. Design level components ensure the reuse of the design idea and efficiency and correctness of design work, whereas implementation level components ensure the reuse of code and the efficiency and correctness of building a software system.

1.2. Component Representation Scheme

We adopt a facet-based component representation scheme (FCRS), providing an easily changeable representation scheme by modifying the underlying facet list. Such a feature meets the challenge posted in the current software development environment where the content in a component library can change rapidly. Although it does not contain much "deep" semantic information, we believe that the facet representation is sufficient for describing components. Like components for other engineering disciplines, software components have common attributes shared by each component, and unique attributes to distinguish individual components from each other. In accord with the above observation, we propose to represent components based on double facet lists. One facet list is used to describe the common attributes and another facet list to describe the unique attributes. We have determined the 10 common facets to uniquely identify a component:

- Component: facet list
- Universal Identifier: string
- Local Identifier: string
- Originator Organization: string
- Domain: string
- Type: string
- Interface: parameter list
- Return Type: string
- Exception: exception list
- Location: string
- Description: facet list

Below is a more detailed explanation of each facet.

Universal Identifier: this facet is used to identify components universally. The value for this facet has to be universally unique, like IP addresses. The scheme to determine universal identifiers should be managed by a centralized organization.

Local Identifier: this facet helps to identify local components. The value is locally unique, but does not have to be universally unique.

Originator Organization: this facet indicates who the originator of the components is. The value has an URL-like format, xxx.yyy.zzz..

Domain: this facet specifies in which application domain the component is working. Possible values could be “finance”, “education”, “medical” and so on. This facet is used to distinguish possible similar components.

Type: this is used to distinguish a component from other components within the same domain by its functionality. The value could be “design pattern”, “framework”, “stack”, “queue” and so on.

Interface: this facet specifies how to interact with the component. For an implementation component, it could be a list of parameters. For a design component, it could be a command used to access the content of the component.

Return type: return type is used to specify what the system will get when the interaction with the component is finished. It is used together with the interface to specify the action of a component. These two facets are important during the composition of components.

Exception list: this is for implementation components only. We use exception lists to define the pre and post-state conditions.

Location: location specifies where a component is. In order to distinguish the address of a library from that of a WWW site, a location-independent naming system is needed. In their paper, Browne *et al* [BRO95], gives a complete discussion of such a naming system.

Description: It contains a pointer to another facet list in which the characteristics of a component are recorded. The facets in this facet list are determined by the type of the component. We believe that when the component technology becomes mature, each component will be classified into one particular class, and components in the same class have certain characteristics to differentiate themselves. In the above representation scheme, the facet of description is the most important one since it uniquely identifies the software component.

2. Component Identification

This section describes the process involved in identifying the characteristics of components. This is a semi-automatic process also suitable for abnormal circumstances, which may require human intervention in recognizing the relevant characteristics of components.

2.1. Ontology

An ontology provides an explicit formal specification for the terms used in a particular domain, and identifies relations among these terms (Gruber, 1993). Such an ontology can be used as a common framework among various parties interested in the domain. It identifies and formalizes the underlying structure of the information and knowledge about the domain. An ontology is a graph whose nodes represent the concepts or objects of a domain, and the edges indicate relationships between concepts. Usually this graph is structured around a hierarchical “backbone” similar to the class/subclass relationship in object-oriented programming. It is not a strict tree, however, and links can exist between any concepts in the graph. Due to the formalization, it can be represented and to some degree interpreted by machines, and enables the formal analysis of the domain. This allows an automated or computer-aided extraction and aggregation of knowledge from different sources and possibly in different formats (as long as the formats can be mapped to the ontology).

To a certain extent, ontologies can mirror class hierarchies, objects, relation, properties, and methods used in software development. The latter, however, usually reflect the perspective of software developers, whereas ontologies concentrate on aspects of the domain that are visible to all parties interested in a particular domain, particularly users of software applications.

From an engineering perspective, ontologies can be very helpful with the reuse of domain knowledge, and for the separation of domain knowledge and software code that performs operations on that knowledge. In our proposed framework, ontology is used in both the signature extraction engine and the facet categorization engine.

2.2. Component Documentation

In component libraries, valuable knowledge such as features, locations and interface definitions, are encapsulated in unstructured or semi-structured documents. We denote such documents as component documentation. Examples of component documentation include source code, software requirement documents, software design documents, bug reports, and technical reports.

2.3. Signature Extraction Engine

In our framework, a signature refers to a set of keywords that links a component to a concept within the ontology. For example, the signature *EDUCATION* may contain the keywords education, training, and learning, whereas the signature *FINANCE* may likewise contain the keywords accounting, tax-return, and commerce. The ontology enables the computer to correlate the occurrence of the words in the component documentation with the respective meaning in the user context (Pan, 2002).

The objective of the signature extraction engine is to construct meaningful signatures according to the ontology from the component documentation. Each time a signature is found, an instance of the signature is created. For example, if the component documentation contains twenty occurrences of the word tax-return, twenty instances of the signature *FINANCE* will be instantiated and stored in the signature knowledge-base. The signature knowledge-base forms the input for the facet categorization engine.

2.4. Facet Categorization Engine

A facet category is defined as a grouping of signatures having the same perspective. For example, the signatures *EDUCATION* and *FINANCE* are grouped in the facet category *DOMAIN*. An ontology defines the facet categories; each category corresponds to a facet in the FCRS. One benefit is that it is possible to extend the ontology dynamically. For example, a new facet category *SPECIFIC-DOMAIN* containing the signatures *EDUCATION* and *FINANCE* can be added to the ontology in run-time, without re-compiling the source code of the facet categorization engine. The resulting facet categories serve as input for the respective neural associative memories.

3. Component Retrieval

This section describes the component retrieval method, as well as the techniques employed in order to facilitate a fast selection of components that best match a given set of criteria for the desired component.

3.1. Neural Associative Memories

A neural associative memory Anderson, 1972; Kohonen, 1972; Palm, 1980) is a single-layered neural network that maps a set of input patterns $X=\{x^1, \ldots, x^m\}$ into a set of output patterns $Y = \{y^1, \ldots, y^m\}$, where each pair $(x^k, y^k) \in \mathrm{IR}^n \times \mathrm{IR}^m$. The associative memory remembers a set, $S = \{(x^k, y^k): k=1, \ldots, M\}$, of mappings. When a new input x is presented to the network, the corresponding output y is calculated by a mapping $y = x\mathrm{W}$, where W is referred to as the synaptic connectivity matrix. During the

learning stage, each pair $(x^k, y^k) \in S$ is presented to the associative memory. This provides a presynaptic and a postsynaptic signal at every synapse. According to these two signals, the synaptic weight is changed. For learning, we use the Hebb rule (Hebb, 1949). It states that if both the neuron at position *i* of the input pattern, x_i, and the neuron at position *j* of output pattern, y_j, are active, the weight of the corresponding synapse, w_{ij}, is increased by 1; otherwise, the weight remains the same. In the retrieval stage, a new input pattern *x* is applied to the input of the network. The input signals are propagated through the synaptic connection w_{ij} to all neurons at the same time. Each neuron j transforms the input signals into its dendritic potential d_j, which is the sum of inputs weighted by the corresponding synaptic strength: $d_j = \Sigma x_i w_{ij.}$ The new activity of neuron *j* is determined by a non-linear operation called threshold detection: $y_j = f_j (d_j - \theta_j)$. The function f_j is called the activation function where θ_j represents the threshold value. This equation is used to determine if the neuron *j* is active or not.

3.2. Retrieval Method

For each component, there are 10 facets to describe the component; in order not to lose generality we suppose there can be at most n facets to represent the component. Let the facets be denoted R_1, R_2, …, R_n. Each facet R_i is a set of finite values, $R_i=\{V_{ij}: j = 1 \text{ to } N_i\}$ where N_i is the number of possible values for facet R_i. The space for the library then is $R = R_1 \times R_2 \times \ldots \times R_n$, where $R_i = \{V_{ij} : j = 1, \ldots, N_i\}$. Let φ denote a component and d_φ denote the representation of a component. Then $d_\varphi = (U_1, U_2, U_3, \ldots, U_n)$ where $Ui \subseteq Ri$. Note that d_φ is a facet list. Let L denote the relevancy between two components. L is defined as follows:

$$L(d_\varphi, d_{\varphi'}) = L((U_1, U_2, \ldots, U_n), (U_1', U_2', \ldots, U_n')) = \Sigma L_f(U_i, U_i') / n \text{ where } i = 1, 2, \ldots, n$$

L_f is called facet relevancy. Note that the relevancy of two components is actually the relevancy between two facet lists. So L is also used to denote the relevancy between two facet lists and called facet list relevancy. It is the sum of all facet relevancies divided by the number of facets. The description facet of a component adopts a facet list to describe a component. The facet list contains several key aspects of the component. We denote the facet list by F and each of the facets by $f_{i,}$ i = 1, …, n; n is the number of facets. For each facet f_i, there is a set of values associated with it, $f_i = \{T_{ik} : k = 1, \ldots, M_i\}$ where M_i is the number of values associated with facet f_i. The space for F then is $F = f_1 \times f_2 \times \ldots \times f_n$. We dedicate one neural associative network N_i to one facet R_i in the representation of components. A binary vector is used to represent the feature space of $R_{i.}$ The dimension of the vector is the same as the number of values for R_i. Thus each bit in the vector represents one value in the feature space. Another binary vector is used to represent the components. We set the number of bits in the binary vector the same as that of components in the component library. One bit in the vector represents one particular component. During the training stage, each facet of the component representations is fed into the dedicated neural associative network. For example, in our representation framework, there are 10 facets, so we use 9 neural associative memories to remember the facet values of each component, except for the facet “Description”. For the description facet, we need more neural associative memories since it is again a facet list. For each facet, we use an n-dimensional vector to denote the feature space assuming that the components have n unique values. After all the training component representations are fed into the associative memory, the synaptic connectivity matrix W is constructed. During the retrieval stage, the desired component representation is broken down into facets. The value for each facet is fed into its dedicated associative memory to recall the components that have the same value for this facet. After one processing step, all the components having this value will be recalled. Note that if the number of searched values is larger than one, the components with at least one value in the look-up list are all recalled. One component may partially match a desired component. This unique feature of associative memory gives us a way to determine how close the retrieved components are to the query.

4. Implementation

Based on the framework outlined above, a system has been designed and implemented for the retrieval of components from a library. A user (client) sends a request to the server looking for a particular component. The client interface can be a standalone GUI or a Web browser. The server gets the request, parses it, converts the necessary information into a query and sends it to the component registry. The registry uses its neural associative memories to search all the representations of components it contains. When a matching component is found, either exact or approximate, it sends the universal identifier or local identifier of the retrieved component to the server. The server sends the representation of the component back to the user. If the user is satisfied with the search result, he sends the request again to the server to retrieve the component by presenting the unique identifier to the server. The server then queries the component library to get the component. The component library performs routine updates, such as adding new components, deleting old components and replacing old versions of components by new versions. It also reports the changes to the registry so that the registry keeps the most up-to-date representation of the components. Every time the state of the component library changes, the component identification process is invoked to adjust the contents of the neural associative memory. Many of these maintenance operations can be performed off-line during low system usage times. Figure 1 depicts the overall system illustration.

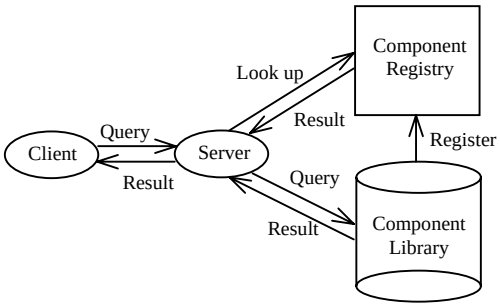

Fig. 1 System illustration.

4.1. Implementation Issues

According to the above analysis, the size of each neural associative memory is O(m2). Based on this analysis, we decided to carefully design our implementation of the associative memories so that it fully utilizes the memory space of a computer. Recall that the input vector, output vector and synaptic connectivity matrix are represented as binary data types. When 0 or 1 is stored in the computer, it is usually treated as an integer, and thus occupies 32 bits or even 64 bits in memory. In order to save computer memory, we want to represent 0 and 1 by one bit instead of one integer. So we compress every 32(64) bits into one integer and use that integer to represent these 32(64) bits in the matrix. Every time we perform an operation on the matrix, we decompress the needed part and compress it back when the operation is done. This way, we sacrifice some execution time for compression and decompression, but we utilize memory much more effectively.

4.2. Extensions

One of the main limitations of our framework lies in the way neural associative memories work: they calculate the similarity of stored items according to the presence or absence of certain pre-defined, simple criteria. The internal structure of components, for example, or other semantic information contained in the specification, is beyond the representational power of these memories. There are some

approaches under investigation that combine the fast access of such neural networks with the higher representational power of recurrent networks (Kurfess, 1999). These networks essentially compress structural information into a very compact representation, which then allows fast operations like the approximate matching of the structure of two graphs. This could be used to specify important structural characteristics of a desired component, and to perform a fast search for the most suitable candidate components.

5. Conclusions

In this paper, we proposed a content-based framework for component libraries and an efficient algorithm using neural associative memories to retrieve components based on the framework. The advantage of this algorithm is a single-step process for each neural network to determine the matching components, and each neural network can work independently. While a prototypical implementation showed the feasibility of the overall approach, some implementation issues like memory utilization had to be treated with special care in order to ensure usability of the system. In addition, the framework is not immediately suitable for the fully automated construction of systems since the matching method only relies on specific features of components, and cannot capture their full semantics. If this is desired, our methods may be used to provide a fast selection of candidates, which then can be examined more carefully for suitability according to their full specification.

6. References

Anderson, J., 1972, "A Simple Neural Network Generating an Interactive Memory," Mathematical Bioscience, Vol. 14, pp. 197-220.

Browne S., Dongarra, J., Green, S., and Moore K., 1995, "Location-Independent Naming for Virtual Distributed Software Repositories," in Proceedings of the 17th international conference on software engineering on Symposium on software reusability, pp. 179-185.

Dewey, M., 1979, "Decimal Classification and Relative Indexing," 19th, ed., Forest Press Inc., Albany, N.Y.

Hebb, D., 1949, The Organization of Behavior, Wiley, New York.

Gruber, T. R., 1993 "A Translation Approach to Portable Ontology Specification" Knowledge Acquisition 5: pp. 199-220.

Jeng, J., and Cheng, B., 1994, "A Formal Approach to Reusing More General Components," in Proceedings of IEEE 9th Knowledge-Based Software Engineering Conference, pp. 90-97, Monterey, Calif.

Kohonen, K., 1972, "Correlation Matrix Memory," IEEE Trans. on Computer, Vol. 21, No. 4, pp. 353-359.

Kurfess, F., 1998, "Component-Based Knowledge Management," Technical Report, New Jersey Institute of Technology.

Kurfess, F., 1999,, "Neural Networks and Structured Knowledge," special issue of Journal of Applied Intelligence, Vol. 11, No. 1, and Vol. 12, No. 1/2.

Latour, L., and Johnson, E., 1988, "SEER: A Graphical Retrieval System for Reusable Ada Software Modules," in Proceedings of 3rd International IEEE Conference of Ada Applications and Environments.

Maarek, Y., Berry, D., and Kaiser, G. , 1991, "An Information Retrieval Approach for Automatically Constructing Software Libraries," IEEE Trans. on Software Engineering, Vol. 17, No. 8, pp. 800-813.

McIlroy, D., 1969,"Mass-produced software components," in Proceedings of the 1968 and 1969 NATO Conferences, pp. 88-98.

Mili, R., Mili, A., and Mittermeir, R. , 1997, "Storing and Retrieving Software Components: A Refinement Based System," IEEE Trans. on Software Engineering, Vol. 23, No. 7, pp. 445-460.

Ostertag, E., Hendler, J., Prieto-Diaz, R., and Braun, C., 1992, "Computing Similarity in a Reuse Library System: An AI-Based Approach," ACM Trans. on Software Engineering and Methodology, Vol. 1, No. 3, pp. 205-228.

Palm, G., 1980, On Associative Memory, Biol. Cybernetics, Vol. 36, pp. 19-31

Prieto-Diaz, R., 1985, “A Software Classification Scheme,” Doctoral Dissertation, University of California, Irvine.

Prieto-Diaz, R., 1991, “Implementing Faceted Classification For Software Reuse,” Communication of ACM, Vol. 35, No. 5, pp. 89-97.

Pan, X.S., 2002 “A context-based free text interpreter,” California Polytechnic State University San Luis Obispo Master’s Thesis - Computer Science Department.

Sametinger, J., 1997, Software Engineering with Reusable Components, Springer Verlag.

Solderitsch, J., Wallnau, K., and Thalhamer, J.,1989, “Constructing domain-specific Ada reuse libraries,” in Proceedings of 7th Annual National Conference Ada Technology.

Szyperski, C., 1997, Component Software: Beyond Object-Oriented Programming, Addison-Wesley, New York.

Tang, Y., 1998, “A Methodology for Component-Based System Integration,” Doctoral Dissertation, New Jersey Inst. of Technology.

Zaremski, A. and Wing, J., 1995, “Signature Matching: A Tool for Using Software Libraries,” ACM Trans. on Software Engineering and Methodology, Vol. 4,No.2, pp. 146-170.