International Journal of Software Engineering
and Knowledge Engineering
Vol. 16, No. 4 (2006) 523–552
© World Scientific Publishing Company

Int. J. Soft. Eng. Knowl. Eng. 2006.16:523-552. Downloaded from www.worldscientific.com
by UNIVESITY OF TEXAS AT DALLAS on 01/30/14. For personal use only.

A REPOSITORY FOR COMPONENT-BASED EMBEDDED SOFTWARE DEVELOPMENT

TONG GAO, HUI MA, I-LING YEN*, LATIFUR KHAN and FAROKH BASTANI

Department of Computer Science, MS EC-31, Box 830688,
University of Texas at Dallas, Richardson, TX 75083-0688, USA
**ilyen@utdallas.edu*

Received 29 March 2004
Revised 12 July 2005
Accepted 20 September 2005

The rapid growth in the demand for embedded systems and the increased complexity of embedded software pose an urgent need for advanced embedded software development techniques. Software technology is shifting toward semi-automated code generation and integration of systems from components. Component-based development (CBD) techniques can significantly reduce the time and cost for developing software systems. Furthermore, effective component retrieval is a fundamental issue in CBD. In this paper, we address the issues in designing software repositories for embedded software components. We develop an On-line Repository for Embedded Software (ORES) to facilitate component management and retrieval. ORES uses an ontology-based approach to facilitate repository browsing and effective search. To allow easy browsing of ORES, we analyze the typical ontology relations for software components and develop a Merging and Echoing technique to convert the ontology into a hierarchy suitable for browsing, but without the loss of any critical semantic information contained in the ontology. We also develop an algorithm for grouping search results based on the ontology. Thus, we can display search result groups to avoid having to display a large number of search results or having to prune the results and risk reducing the recall factor. Another important aspect in embedded software is the set of nonfunctional requirements and properties. In ORES, we develop an XML-based specification method to capture nonfunctional properties as well as functional characteristics of components and enable retrieval of relevant components based on these specifications.

Keywords: Component-based software engineering; component repository; embedded software; ontology; browsing and search; nonfunctional requirements.

Introduction

To enhance the productivity while developing complex applications, software technology is rapidly shifting away from low-level programming issues to automated

*Corresponding author.

code generation and integration of systems from components by using either Commercial-Off-The-Shelf (COTS) components or specially developed in-house components [1, 2]. The component-based development (CBD) approach can significantly reduce software development time and cost. Among various issues, component retrieval is the key to the success of CBD techniques. The retrieval process involves matching the desired functionality and making sure that the component satisfies required nonfunctional properties such as timing requirements and resource constraints.

Over the past decade, component retrieval has been studied extensively [6, 34]. Desirable retrieval techniques should yield high precision and recall [13]. Let I be the set of components that should be returned for a retrieval query, and let R be the set of components actually returned. Precision can be defined as $\frac{||R \cap I||}{||R||}$, that requires the retrieval algorithm to return only the relevant components. Recall can also be defined as $\frac{||R \cap I||}{||I||}$, that requires the retrieval algorithm not to miss relevant components [13].

Formal methods have been used [17, 34] to achieve better precision and recall in component retrieval. There are two major approaches along this direction. In syntax-based retrieval, component selection is based on matching the signatures of the operations, such as input/output parameter types [13, 34]. Since syntax based retrieval does not provide a complete behavior description, it is not suitable for partially specified retrievals. The semantics-based approach specifies a component by its behavior. Generally, formal methods are used for behavior specification [6, 34]. Theorem proving or rule-based reasoning techniques can be used to determine equivalence or similarity of component behavior [19]. These are elegant solutions; however, they are difficult to use due to the low-level granularity of formal specifications. For example, the distinction between a data compression component and a data encryption component is conceptually very clear but a lot of details may be required in the formal specifications to define the corresponding component behaviors clearly. Such specifications can also result in a substantial semantic gap between the user view of a component and its formal specification. Thus, it is difficult for most practitioners to concisely and uniquely specify components using formal methods.

We believe that modeling a software component repository based on information technologies, such as the concept of ontology and search techniques, can reduce the semantic gap between the conceptual structure of components in the repository and the users' view and, hence, can enable easy and effective component retrieval. We use an ontology to organize the components in the repository. Ontology is a collection of nodes and their relationships [5, 10], which collectively provides an abstract view of a certain application domain. It can provide a similar sub-structure as that in enumerated classification and, thus, facilitates effective browsing. Also, we design ontology-based search models and algorithms so that the search scheme fully exploits the meta-information offered in the ontology to improve the level of precision and recall.

Int. J. Soft. Eng. Knowl. Eng. 2006.16:523-552. Downloaded from www.worldscientific.com
by UNIVESITY OF TEXAS AT DALLAS on 01/30/14. For personal use only.

Int. J. Soft. Eng. Knowl. Eng. 2006.16:523-552. Downloaded from www.worldscientific.com
by UNIVESITY OF TEXAS AT DALLAS on 01/30/14. For personal use only.

Most component retrieval research has focused on functionality match. However, for some applications, such as embedded software systems, component retrieval generally involves consideration of nonfunctional requirements (NFRs), such as timing constraints, memory constraints, security requirements, etc. Thus, the design of the component repository and the retrieval techniques should also consider NFRs [8, 27, 29].

In this paper, we present the design and implementation of an On-line Repository for Embedded Software (ORES). ORES has two major design goals, namely, effective component retrieval and nonfunctional property capture. We retrieve components by browsing and searching. In ORES, an ontology is used to capture various types of relationships among software components. We analyze the required ontology features for different types of software components and develop ontology construction techniques accordingly. The ontology in traditional information systems is constructed based on the concepts and knowledge structure. This can be applied to software component repositories as well. However, software components are also naturally grouped by certain "syntactical" boundaries, such as packages, programs, classes, etc. Additional relationships, such as inheritance and use, should also be considered in the ontology. Thus, the ontology for software components tends to have complex multiple views which can make ontology-based navigation and browsing difficult. We develop techniques to merge multiple views in software component ontology and build a single hierarchy that facilitates easy navigation and browsing. The ontology also helps the users in understanding the components. The search function in ORES makes use of the ontology information to group search results. When a large number of search results are produced, we group them according to the hierarchical structure embedded in the ontology and display only the representative nodes of a group. Users can then guide the search process interactively. Finally, we develop an XML-based model to effectively capture the general information as well as nonfunctional attributes for each component in ORES. We also develop tools to effectively present the nonfunctional properties and facilitate component selection.

The rest of the paper is organized as follow: In the next section, we present the major design concepts for ORES, focusing on the ontology design that facilitates navigation and search. In Sec. 3, the ontology-based search scheme is presented. The model for component description, especially the description of nonfunctional properties of components, is introduced in Sec. 4. In Sec. 5, we present the implementation issues for ORES and introduce the tools associated with ORES. In Sec. 6 we summarize the paper and identify some future research directions.

1. ORES Ontology and Ontology Navigation

Existing component repositories generally consider one relationship between the nodes. For example, taxonomy based repositories organize the components based on their functional behaviors, but do not consider the correlations due to other factors, such as calling sequences and package structures. Consider the example

of a voice-over-IP system. The sender needs to *encode* the voice stream and the receiver needs to *decode* the stream. Conventional ontology design may capture the relationship between different versions of "encoder" or "decoder" functions, but not the correlation between a specific pair of "encoder" and "decoder" functions from the same package that have to be invoked in pairs. Conversely, Java libraries are structured based on packages, classes, methods, etc. Similar functionalities are not directly associated with each other in Java libraries. For example, in the Java socket library, "DatagramSocket" and "ServerSocket" are different classes, but they contain similar functions such as "get/set address", "send/receive message", etc. However, these correlations based on functional behaviors are not maintained in Java.

To offer better representations of the correlations among components, it is necessary to provide multiple views in a software component repository. Two major views include the "semantic" view, which links components with similar functional behaviors together and labels them, and the "syntactical" view, which captures the boundaries of software packages, classes, etc. In the ORES repository, we use an ontology to capture multiple relationships among the nodes. In the following subsections, we will discuss the details of the ORES ontology and use examples to illustrate the concept. We use UML [25] to specify the ontology. Most of the nodes and relations in the ORES ontology can be described using standard UML elements and relations. Also, UML has three extensibility mechanisms of which the stereotype mechanism allows controlled extension of metamodel classes. Some of the types and relations in the ORES ontology (those that cannot be specified by standard UML) are represented using stereotypes.

1.1. *Syntactical view of ORES ontology*

First, we consider the syntactical view of the ontology, which structures the components in a tree. Each node in the repository ontology has a type, which can be domain, package, class behavior, class, function behavior, or function. At the highest level, we have a root node which represents the overall domain of the entire repository. Various children domain nodes are **specializations** of their parent domains [25]. All the domain nodes form a high level hierarchy. A package node is associated with its parent domain node with the **realization** relation [25]. Note that a package node can be a "realization" of a domain node at any level. In general, a software package consists of a number of classes, where each class is a program unit that encapsulates certain abstract concepts or behaviors together with some state information that is accessible within the class. Also, a group of functions is implemented in a class to access the encapsulated state information and/or achieve certain goals of the class. However, in a large package, there may be hundreds of classes. We further group the classes into a hierarchy based on their behavior and "class-behavior" nodes are added to the sub-hierarchy. Similarly, a large class can have a large number of functions and we use "function-behavior" nodes to organize

Int. J. Soft. Eng. Knowl. Eng. 2006.16:523-552. Downloaded from www.worldscientific.com
by UNIVESITY OF TEXAS AT DALLAS on 01/30/14. For personal use only.

Int. J. Soft. Eng. Knowl. Eng. 2006.16:523-552. Downloaded from www.worldscientific.com
by UNIVESITY OF TEXAS AT DALLAS on 01/30/14. For personal use only.

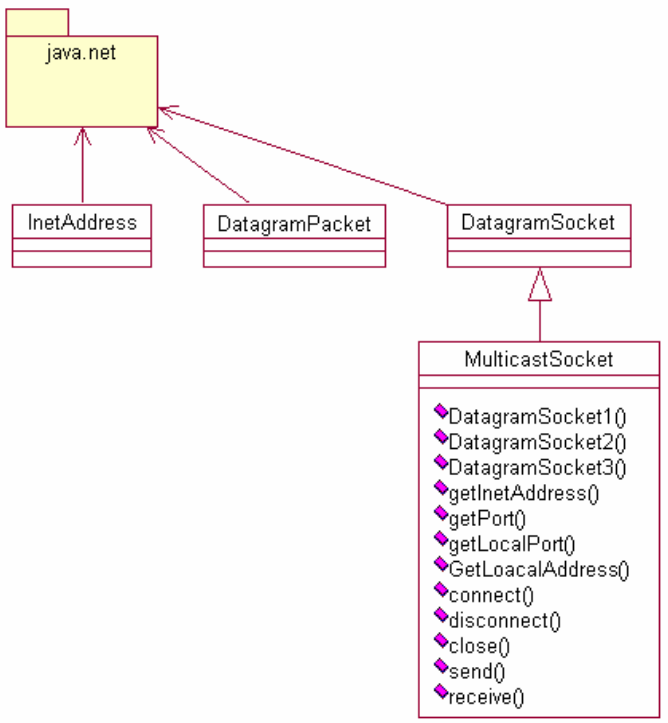

Fig. 1. The hierarchy in package java.net.

these functions in a hierarchy. The "class-behavior" nodes are **associated** with a package and the "function-behavior" nodes are **associated** with a class [25]. One or more actual classes or functions are associated with a class behavior or a function behavior with ther **realization** relation, respectively [25]. Adding description nodes for grouping and describing functional behaviors of regular nodes results in an ontology that is similar to the Smalltalk protocol [26]. In Smalltalk, a protocol is a collection of selected methods of a single class, associated with a keyword phrase. In ORES, a "function-behavior" node groups a collection of functions and describes their common behavior. But a "function-behavior" node may cross class boundaries. Also, we apply the same concept not only to functions, but also to other types of nodes (such as classes and packages).

Here, we illustrate the ontology construction in ORES using the Java Networking package. First, we show the original program hierarchy in **java.net** in Fig. 1. The java.net package contains several classes. Each class contains a set of constructors and a set of methods. Note that only a part of the package hierarchy is shown in Fig. 1.

We further consider the behavior of the classes and build a hierarchy of class behaviors. Here, we can identify three types of behaviors of the classes, namely "reliable-communication", "unreliable-communication", and "multicast-

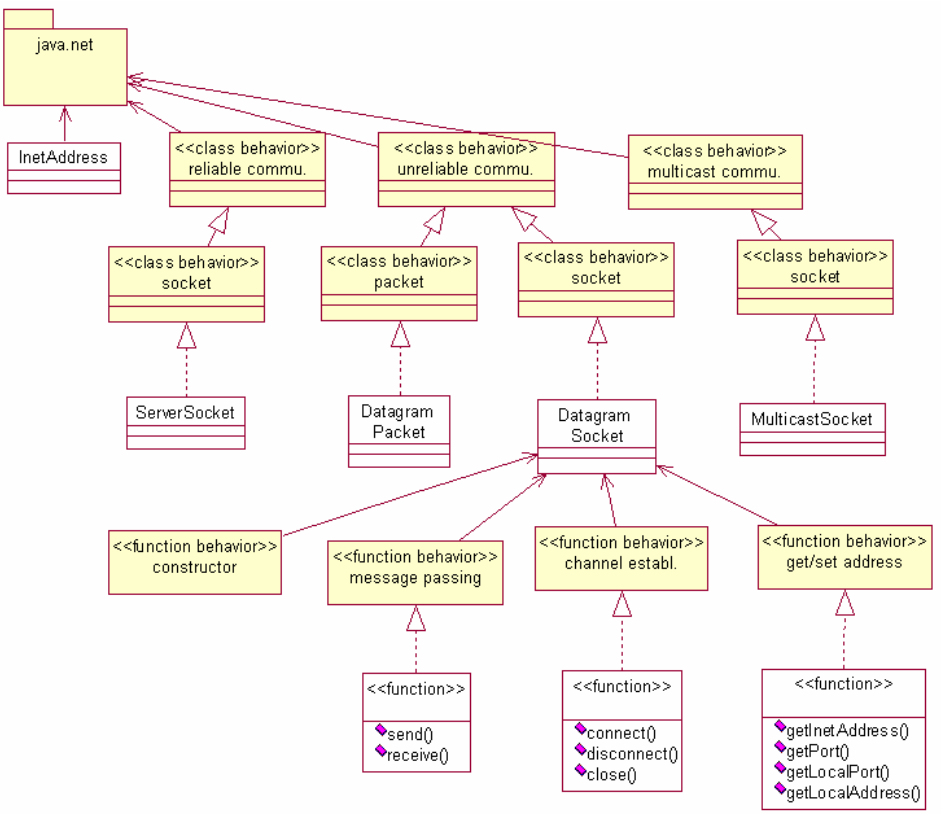

Fig. 2. The syntactical view of the ORES ontology for package java.net.

communication". Under "unreliable-communication", we further identify two class behaviors, "packet" and "socket". Under "unreliable-communication.packet", we have the realization class "DatagramPacket". Under "unreliable-communication. socket", we have the realization class "DatagramSocket". Within each class, we identify function behaviors. In the class DatagramSocket, we consider function behaviors "constructor", "get/set-socket-address-information", "channel-establishment", and "message-passing". Figure 2 shows the partial ontology in ORES for the java.net package.

1.2. *Multiple views of ORES ontology*

Let H^P denote the syntactical view of ORES ontology. Besides H^P, the ontology maintains other relations among the nodes. Consider two packages, A and B, implementing similar functions. Let H_A^P and H_B^P denote the syntactical hierarchy within packages A and B, where $H_X^P = (V_X, E_X)$ where V_X is the set of nodes and E_X the set of edges in the ontology within package X. Note that H_X^P, for some X, is a subset of H^P. Furthermore, we define $V^X = \{V_X^i | 1 \leq i \leq s_X\}$, where

Int. J. Soft. Eng. Knowl. Eng. 2006.16:523-552. Downloaded from www.worldscientific.com
by UNIVESITY OF TEXAS AT DALLAS on 01/30/14. For personal use only.

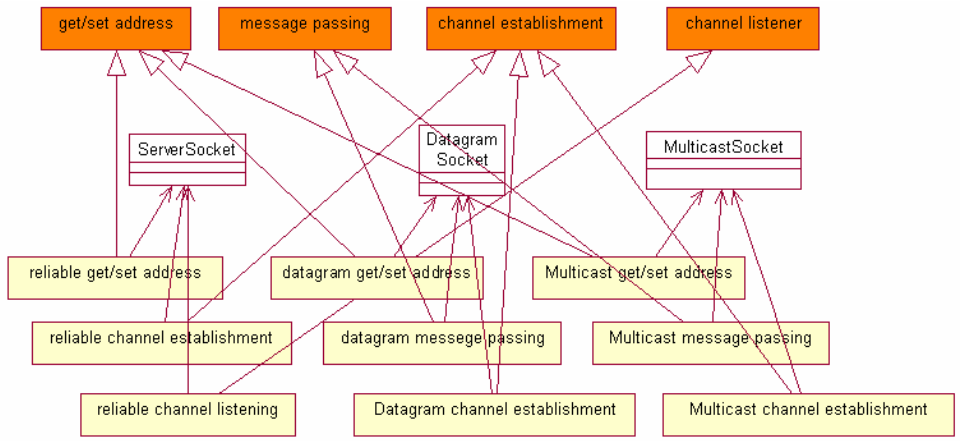

Fig. 3. Multiple views in the ontology for java.net.

s_X is the size of V_X, and $E_X = \{E_X^{ij} |$ for some i and $j, 1 \leq i, j \leq s_X$, and E_X^{ij} is the edge from V_X^i to $V_X^j\}$. Since A and B are similar packages, we can find some nodes in V_A and V_B that have similar behaviors. Let V_A^i and V_B^i be the two nodes in A and B that have similar behaviors. It is desirable to create a node C^i to represent the common behaviors of V_A^i and V_B^i and C^i has V_A^i and V_B^i as its children nodes. C^i will be a node in a different hierarchy H^B that organizes the software according to their behaviors and semantics without considering software package boundaries. Thus, the ontology consists of multiple views, H^P and H^B. Figure 3 shows the multiple views for java.net. From the semantic view H^B, we can define various behaviors, such as "channel-establishment", "message-passing", etc. The node "message-passing" has two child nodes realizing the specific behavior, including "datagram-message-passing" and "multi-cast-message-passing".

1.3. *Merging and Echoing*

The browsing function is provided by ORES to allow users to navigate the ontology and browse the components based on a structured view of the components in the repository. However, the multiple views of the ontology can make browsing confusing. Each type of relationships (such as syntactical and semantic relationships) forms a view in the ontology. If we follow one view for navigation, then the user may lose the other views. For example, navigating class hierarchies is like following the syntactical view. If a user is not familiar with the package that he or she is browsing (say package A) but is familiar with another package (say package B) that has similar functionalities as A, then the knowledge of B may not be helpful due to the lack of semantic associations. However, it is very difficult to navigate through the repository if we try to follow two or more types of relationships since it is no longer a tree structure. In this case, from each node, the user or the system

Int. J. Soft. Eng. Knowl. Eng. 2006.16:523-552. Downloaded from www.worldscientific.com
by UNIVESITY OF TEXAS AT DALLAS on 01/30/14. For personal use only.

needs to decide which link to follow. Thus, we merge multiple views to facilitate navigation while retaining the benefits of multiple views.

In our approach, we use a Merging and Echoing technique to construct a single hierarchy from multiple views while still retaining the characteristics of multiple views. In the Merging and Echoing technique, we mainly consider the semantic and syntactical views. Several subtrees in the syntactical view may be correlated due to their behavioral similarities and linked together by semantic relations. Similar subtrees are merged by taking the "union" of the behaviors and echoed to replace the original subtrees. By doing so, we construct a single hierarchy based on the syntactical relations and retain the semantic relations by keeping the merged structure.

Consider packages A and B (following the example given above). Assume that nodes V_A^i and V_B^i, $1 \leq i \leq s_c$, are similar. We first merge H_A and H_B to obtain a common hierarchy $H_{new} = (V_{new}, E_{new})$, where $V_{new} = \{V_A^i, V_B^j | 1 \leq i \leq s_A, s_c < j \leq s_B\}$, $E_{new} = E_A \cup \{E_{new}^{ij} |$ for i and j, $s_c < i \leq s_B\}$. The merge operator is similar to union. It takes the union of H_A and H_B, except that nodes in H_A and H_B may be "similar" instead of the same; these similar nodes are considered "the same" during merging. After merging, the new hierarchy H_{new} is echoed to replace the original H_A and H_B. This merged and echoed hierarchy provides the virtual semantic links that implicitly correlate the corresponding nodes in several packages. The echoing scheme can be used for multiple packages and applied to any sub-hierarchy at any level.

Figure 4 shows the final ontology for the java.net package. Nodes that belong to echoed hierarchies are shaded. Since the class behaviors "reliable-communication", "unreliable-communication", and "multicast-communication" are similar, we can use the Merging and Echoing scheme to relate their internal ontologies. Even though in the original java.net package there are no explicit "packet" classes for "multicast-communication" and "reliable-communication", the "class-behavior" node, "packet", is still added in each case as a result of ontology echoing. Also, the "realization" link is given to clearly define the actual class that realizes the class behavior. Similarly, we consider all socket classes together for ontology echoing. After echoing, each socket class defines five function behaviors, including "constructor", "channel-establishment", "channel-listening", "message-passing", and "get/set-socket-address-information". Since IOStream is used for message I/O in ServerSocket, the "message-passing" group is empty. Likewise, the "channel-listener" group in DatagramSocket and MulticastSocket are empty. Though some echoed nodes are empty, they do help correlate similar entities in a uniform way. In Fig. 4, we can also see additional relations among components, namely, the inheritance and use relations. For example, MulticastSocket is inherited from DatagramSocket. Also, "packet" in MulticastPacket directly uses DatagramPacket. However, inheritance and use relations are not considered during browsing.

As we can see, the ORES ontology provides a structure that facilitates repository navigation and browsing. With the categorization within the packages and classes,

Int. J. Soft. Eng. Knowl. Eng. 2006.16:523-552. Downloaded from www.worldscientific.com
by UNIVESITY OF TEXAS AT DALLAS on 01/30/14. For personal use only.

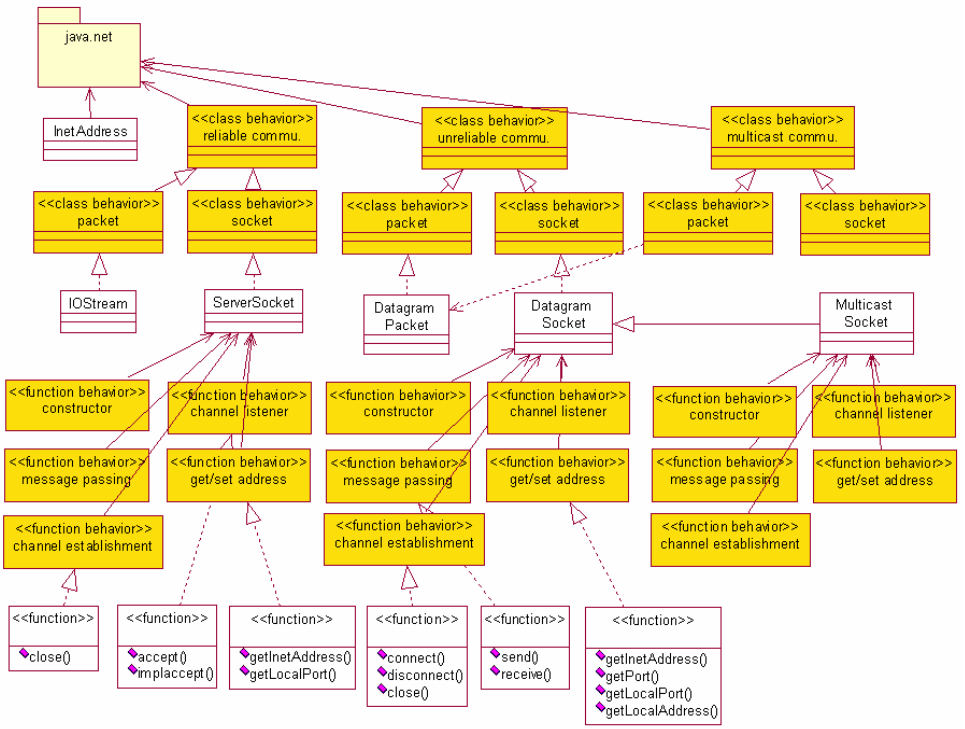

Fig. 4. The ontology of the java.net package in ORES.

relations among components are better captured. With the Merging and Echoing technique, users may be able to more easily understand the relationships among various entities and compare them conveniently. Also, once users understand the components in one group, it is easy to extend the knowledge to other echoed groups.

2. Component Descriptions

A component is a code segment that has a clear interface and a well defined behavior. A component can be a package, a class, or a function. Components can be nested, for example, classes are nested inside a package and functions are contained within a class. Several attributes are required to describe a component and different components may require different attributes to explicitly describe their characteristics. ORES uses the object-oriented model to define the information attributes hierarchy. At the repository root node, we define a basic set of information attributes. These attributes are inherited by all nodes in the repository. A child node inherits the information attributes defined by its parent node and it can modify the definition by adding additional attributes or removing some of them.

Note that ORES is intended for general software repositories with special emphasis on embedded software components. Thus, we specifically address the description model for the nonfunctional properties.

Int. J. Soft. Eng. Knowl. Eng. 2006.16:523-552. Downloaded from www.worldscientific.com
by UNIVESITY OF TEXAS AT DALLAS on 01/30/14. For personal use only.

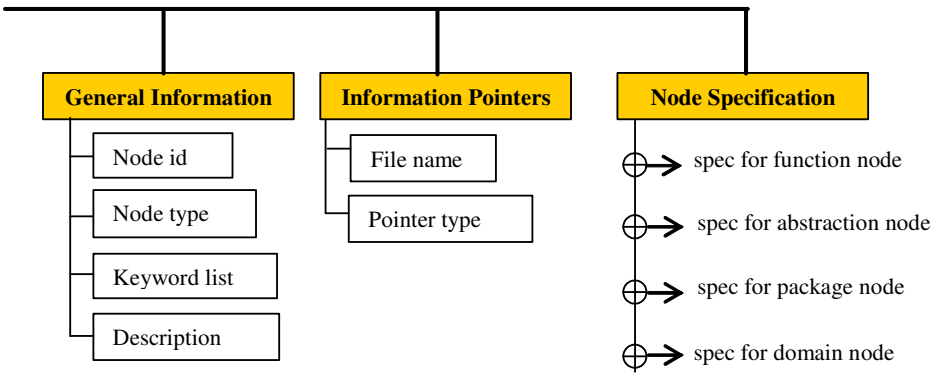

Fig. 5. Basic component description model for the root node.

Int. J. Soft. Eng. Knowl. Eng. 2006.16:523-552. Downloaded from www.worldscientific.com
by UNIVESITY OF TEXAS AT DALLAS on 01/30/14. For personal use only.

2.1. *Basic component description*

Figure 5 illustrates the basic component description model which is specified at the root node and inherited by all nodes in the repository. The information attributes include three categories, namely general information, information pointers, and node specification. In the category "general information", node-id is defined by the path name of the node, which concatenates the names from the root node all the way to the current node. Node-type can be domain, package, class, function, etc. (as defined in Sec. 1.1). The "keywords" field is maintained to facilitate search. A short description of the node is also provided. In the category "information pointers", file name is the file that the node links to. A set of pointers is defined to link to external information files. The pointers are classified into three types, namely, code pointer, document pointer, and web links. Code pointers point to source code and executable files. Document pointers point to documents, such as the user manual, programmer's guide, tutorials, component description, problem/bug report, reliability information, review information, limitations, usage experience report, etc. Web links point to related documents on the web. "Node specification" provides relatively formal specification for a node. For different types of nodes, the specification will be different. For function nodes, the API specification (input, output, exceptions) should be provided. Also, the function type is required for a function node. Function types include constructor, data manipulation functions (such as get-set), data abstraction functions (e.g., add-delete-update), conventional functions (e.g., sort, matrix-operations, etc.), IO functions, and database functions. For class nodes, the class API, such as interface functions and exceptions, and inheritable or externally accessible variables can be provided. For a package node, the package version number, relevant version information, new release information, licensing information, price information, vendor, etc., can be provided. Some package-wide glossary definitions can also be provided. For other node types, such as the domain node, there is no formal specification and the overall description of the node is sufficient.

Int. J. Soft. Eng. Knowl. Eng. 2006.16:523-552. Downloaded from www.worldscientific.com
by UNIVESITY OF TEXAS AT DALLAS on 01/30/14. For personal use only.

2.2. *Description of nonfunctional properties*

Consider a component C. The nonfunctional properties of C can be defined by a set of measurable attributes. Let M denote the set of measurable attributes identified for a component C. $M = \{m_i | \text{for all } i\}$, where m_i is a measurable attribute such as time, memory usage, power consumption, output quality, etc. In general, a component itself does not independently determine the measurements of the attributes in M. Several other factors can impact these measurements. First, the execution platform will greatly impact these measures. For example, the CPU speed, memory type (such as on-chip/off-chip cache, RAM), and OS can greatly impact the execution time. Second, for some components, the input domain can have significant impact on these measures. For example, an input voice stream with a high sampling rate will require a longer processing time and a larger memory size. Finally, time, quality, and resource requirements can impact each other through configurable parameters in the code. In order to correctly measure the component property attributes, we need to record all parameters that can impact the measurement of nonfunctional attributes. We classify these parameters into (i) input parameters, (ii) execution environment parameters, and (iii) configurable program parameters. Let I denote the set of input parameters, E the set of environment parameters, and A the set of configurable program parameters. I, E, and A form an n-dimensional parameter space (where n = total number of parameters in I, E, and A) and each attribute m_i can be measured and plotted against the parameter space, i.e., we can obtain $m_i = f_i(I, E, A)$. In some cases, a simple f_i, where $m_i = f_i(I, E, A)$, can be derived in a straightforward way without any measurements. Also, in some rare situations, f_i may be derived easily from the measured data. In many cases, it may be difficult to obtain the functional representation f_i for some attribute m_i (due to the limited measurements we may be able to obtain). Thus, we allow two forms of expressing the mapping from (I, E, A) to m_i. Either the functional representation f_i or the raw measurement data can be stored in the repository. In the case of the functional representation, we confine it to be polynomial functions and maintain the coefficients. In the case of raw data, we store the values of the parameters and the corresponding measurements for the attributes in the repository. With sufficient amount of measurements, we can use interpolation techniques to predict the data for the entire parameter space; otherwise, the available data will be used for various analysis without interpolation.

We have identified some typical types of measurable nonfunctional attributes for embedded components (as shown in Fig. 6), namely, time, resource requirements, and output quality. Within each category, many individual attributes can be identified. Some sample measurable attributes considered currently in ORES are given in the following.

Output quality measurement is highly component dependent. For example, the quality of an echo canceller component is generally measured by Terminal Coupling Loss (TCL). Some standard attributes include precision, accuracy, integrity, and confidentiality.

- Time measurements (average and worst case values)
  - — CPU cycles
  - — I/O time
  - — communication latency
  - — execution time (in terms of real-time)
- Resource requirements (average and worst case values)
  - — memory requirements
    - – source: by program/data
    - – persistent: stays after completing the execution of the component
    - – volatile: returned to the system after finishing execution
    - – static: fixed amount of memory allocated at the beginning of the execution
    - – dynamic: varying amount of memory allocated at run time
  - — disk requirements
  - — power consumption
  - — communication channel capacity requirements
- Output quality (highly component dependent)
  - — precision, accuracy, integrity, confidentiality

Fig. 6. Typical types of measurable nonfunctional attributes for embedded components.

```
<!ELEMENT Information-Attributes (General, Pointers, Specification, Properties)>
<!ELEMENT General (node-id, node-type, keywords, short-description)>
<!ELEMENT Pointers (source-code-pointer, executable-pointer, tutorial-pointer,
                    document-pointer, additional-links)>
<!ELEMENT Specification (#PCDATA)>
<!ELEMENT Properties (#PCDATA)>
```

Fig. 7. DTD for information attributes definition of ORES root node.

2.3. *XML-based component description*

The descriptions of components in ORES are stored using "Documentation" (standard tagged value) of the appropriate UML standard (or stereotype) elements. The description language is based on XML. Special XML tags are defined for the specification of general information and nonfunctional properties of each component. DTD (document type definition) is used to specify the information attributes. A DTD file is associated with the root node to define the basic information attributes for the repository (see Fig. 7). This is a straightforward realization of the information attributes defined in Sec. 2.1.

When different information attributes are required for a node, the DTD file can be redefined by adding or removing the attributes from the node's parent's DTD. In ORES, the child node that redefines the DTD file actually holds the complete definition (not just the added and removed attributes). Each node maintains a pointer that points to a DTD file that defines the information attributes. This DTD file is maintained by the nearest parent of the node. The DTD file can be

Int. J. Soft. Eng. Knowl. Eng. 2006.16:523-552. Downloaded from www.worldscientific.com
by UNIVESITY OF TEXAS AT DALLAS on 01/30/14. For personal use only.

used to generate the user interface for component description data entry as well as to check the completeness of the component description.

Note that in Fig. 7, the Properties and Specification tags are left as text. This is defined for the root node to guide the basic node description. Different descriptions may be needed for different groups of nodes. Specific DTDs are redefined based on the basic DTD for specific domains, packages, etc. A DTD for the SDF components (as shown in Fig. 11) will be given in Sec. 4 to illustrate extended node descriptions.

Int. J. Soft. Eng. Knowl. Eng. 2006.16:523-552. Downloaded from www.worldscientific.com
by UNIVESITY OF TEXAS AT DALLAS on 01/30/14. For personal use only.

3. Component Search and Ontology-Based Results Grouping

A user may browse the repository to locate a component, but frequently the user may not have extensive knowledge of the ontology and prefers to use the search function. We use conventional search techniques in the information retrieval field to implement the search function [9, 18, 21]. Each node in ORES is described by a concept, which is further defined by a set of keywords. The keywords for each component are obtained by analyzing the component documents and source code. A weight is assigned to each keyword according to its occurrence frequency and location. Let T denote the hierarchy generated by the Merging and Echoing process and $T = (V, E)$, where $V = \{v_i | \text{for all } i\}$ is the set of nodes and E is the set of edges in T. Also, let $v_i \cdot K = \{v_i \cdot K_j | \text{for all } j\}$ denote the set of keys that describes v_i and let $w(v_i \cdot K_j)$ denote the weight of key $v_i \cdot K_j$ for component v_i (note that $v_i \cdot K_j$ is obtained by analyzing the source code and documents associated with v_i). We maintain an inverted list I in memory to keep track of all the keywords from T, where $I = \{K_k | \text{for all } k\}$ and $\forall\, k, \exists\, i, j$, s.t. $K_k = v_i \cdot K_j$. Component links are maintained for each keyword K_k in the inverted list to point to components v_i, for all i, if $K_k = v_i \cdot K_j$. During the search process, keywords given by the user are matched against those in I (various keyword aggregations, such as *and*, *or*, *not*, etc., can be applied). Following the component links from keywords in I, the matching components, which are the search results, are marked on T.

A major problem with keyword-based search is the potential of having a large number of search results. Various methods have been developed to filter the search results [11, 22]. However, search result filtering has the potential of reducing the recall level, i.e., some potentially useful items may be removed. Instead, we use a search results grouping approach to allow the user to direct the search process toward the correct direction to obtain the desired search results. For example, consider searching for a component with a keyword "search". The user may refer to the programs related to "search engines" or "heuristic search algorithms". The user clearly knows which category of components he or she wants to retrieve, but it is difficult for him or her to add additional keywords since it may eliminate some desired results. Clustering techniques have been used to group search results [30]. However, in ORES, the repository ontology provides valuable information that can be used for grouping search results.

3.1. *Ontology-based grouping of search results*

As discussed above, $T = \{V, E\}$ is the tree that represents the main hierarchy of the ontology. Let S_0 denote the set of search results and $S_0 \subseteq V$. Also, let N denote the number of search results, where $N = |S_0|$. At the end of the search process, all nodes in S_0 are marked on T (through the links from K_k in I to v_i in T). If N is large, it is desirable to group the search results in S_0. Basically, we group nodes in a subtree and use the root node of the subtree to represent the group. The goal is to maintain the original ontology while presenting grouped search results. Based on the structure and node types in the ORES ontology, we define a distance measure between the nodes to guide the search result grouping process. Let S_f denote the set of nodes that are selected to be presented to the user. Each node in S_f represents a group of nodes, including itself and its descendants.

First, we discuss various parameters that should be considered for controlling the results grouping mechanism to achieve the desired effect. Most basically, a bound B_N should be chosen to decide when to activate the grouping algorithm. If $N \leq B_N$, then the number of search results is reasonably small and grouping is not needed. B_N can be determined by the user or set to a system default (such as 50). When $N > B_N$, the grouping computation gets activated. Next, we should consider a reasonable number of groups to be presented to the user at time g. Generally, g should be smaller than B_N since it may be harder to go through a list of groups than a list of search results. Also, g can be dynamically determined by N. With a larger N, a larger g can be chosen to reduce the potential for a user of having to go through too many option screens before reaching the desired nodes. Once g is determined, we can correspondingly determine M, the minimum number of nodes that can form a group. On average, $M = N/g$. But we should allow variable group sizes since the search results may distribute over T arbitrarily and it may be reasonable to have more nodes in a group in a dense area on T. Also, with too small a group, it may be better to directly show all the search results instead of presenting the representative node of the group. Thus, a reasonable M can be some value around $N/2g$. Finally, we need to consider the bound D, the maximum distance between the nodes in a group. A larger D value indicates less similarity in the group and, hence, the representative node may not be very indicative for its group. To choose the maximum distance, D, we need to first define the distance between nodes.

Let $d(x, y)$ denote the distance between two nodes x and y, where $x, y \in T$. First, we define the distance between parent-child nodes. Consider nodes p and c, where $p, c \in T$, and p is the immediate parent of c. The distances between various types of p and c are assigned based on heuristics as follows. Note that we have $0 \leq d(p, c) \leq 1$.

1. $d(p, c) = 1.0$ if $p, c \in S_0$ and both p and c are domain nodes
2. $d(p, c) = 0.8$ if $p, c \in S_0$ and p is a package node and c is a "class-behavior" node

Int. J. Soft. Eng. Knowl. Eng. 2006.16:523-552. Downloaded from www.worldscientific.com
by UNIVESITY OF TEXAS AT DALLAS on 01/30/14. For personal use only.

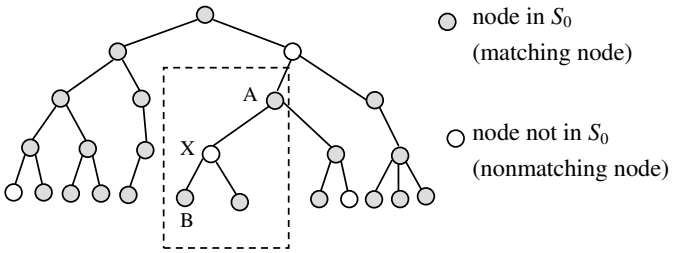

Fig. 8. An example of distance between the nodes in ontology.

Int. J. Soft. Eng. Knowl. Eng. 2006.16:523-552. Downloaded from www.worldscientific.com
by UNIVESITY OF TEXAS AT DALLAS on 01/30/14. For personal use only.

3. $d(p, c) = 0.7$ if $p, c \in S_0$ and both p and c are "class-behavior" nodes
4. $d(p, c) = 0.5$ if $p, c \in S_0$ and p is an class node and c is a "function-behavior" node
5. $d(p, c) = 0.5$ if $p, c \in S_0$ and both p and c are "function-behavior" nodes
6. $d(p, c) = 0.6$ if $p, c \in S_0$ and p is a domain node and c is a package node
7. $d(p, c) = 0.3$ if $p, c \in S_0$ and p is an "class-behavior" node and c is a class node
8. $d(p, c) = 0.2$ if $p, c \in S_0$ and p is a "function-behavior" node and c is a function node

In the case of the realization relation, the child node has the same concept as the parent where the parent node represents the abstract concept and the child node represents the implementation of the concept. Thus, the distance for the realization relation is relatively small, as those given in rules 6, 7, and 8. The similarity of the parent and children nodes in the specialization and association relations is not as much as that in the realization relation. Thus, the distance between the nodes is relatively larger (as given in rules 1 through 5). Also, in a smaller component unit, the relations between nodes get closer. For example, the functions in a class are more closely related than the classes in a package. The distance definitions given above also reflect this property.

In the search for a certain concept (defined by the set of given keywords), a matching node and a nonmatching node are relatively far apart in terms of the concept. We further define two more distances:

9. $d(p, c) = 1.5$ if $p \in S_0 \wedge c \notin S_0$, and
10. $d(p, c) = 1.5$ if $p \notin S_0 \wedge c \in S_0$.

Consider three nodes A, B, and X, where A and B are in S_0 and X is not in S_0, A is the immediate parent of X and X is the immediate parent of B (as shown in Fig. 8). For this instance, though both A and B are in S_0, we can view A and B as being less similar because of the nonmatching node X, so the distance between A and B is greatly enlarged. Hence, if the distance of A and B is defined as normal, this group may include more dissimilar nodes like A and B and, thus, its representative node cannot properly represent the nodes in this group.

Based on the definitions of distances between two adjacent nodes, the distance between two arbitrary nodes x and y in T can be defined in a recursive way as follows.

$d(x, y) = d(x, z) + d(z, y)$, where x is the immediate parent node of z.

We define G as a subgraph of T. The nodes and edges of the subgraph belong to the set of the original graph T. Since two subgraphs cannot have the same nodes, each node can be included in only one group and be represented by other nodes only once. We can also view the subgraph as a continuous connected part of the original graph T. Then, we have the following definition:

$d(x, G) = \max d(x, y), y \in G$, x is the root of G.

So, we need to choose G, such that $d(x, G) < D$, where x is the root of G. The original graph T can be composed of several subgraphs. Each subgraph can choose one node to represent other nodes in the subgraph G which also belongs to S_0. The final search results will be $G \cap S_0$.

The search terminates when one of the following occurs:

(1) The tree traversal reaches the root.
(2) When a node P is encountered such that $d(x, G) > D$, where G is the cluster of selected nodes.

3.2. *An efficient algorithm for grouping search results*

The number of nodes in a large repository can be very high. Thus, the efficiency of the algorithm for grouping the search results is critical. To group search nodes, we can either traverse the entire tree T from the root or start from nodes in S_0 and try to find nearby nodes that are suitable for grouping. Here, we assume that the percentage of matching nodes to the total number of nodes (S_0/T) is relatively small. Thus, to achieve better efficiency, the grouping process starts from the nodes in S_0 to derive the final set S_f which only contains the selected representative nodes to be displayed as the search results.

The algorithm for computing S_f from S_0 consists of two phases. In the first phase, we start from nodes in S_0, traverse up T, and mark the traversed nodes. The maximum distance to traverse from a node in S_0 is limited by D. At the end of the first phase, each node q knows the exact number of its children nodes that are marked (involved in the computation). This number is used in the second phase to prevent q from being processed till all its marked children nodes have been processed. For each node q, when q is first visited, we set $q.mark$ to 1 and update the distance field of q's parent node p. Note that $p.distance$ is the shortest distance of p's immediate children nodes that are marked. It is used to control the traversal and limit the distance to be within D. If node q is already marked, we update $q.num$ which is the number of q's immediate children nodes that are marked. Also,

Int. J. Soft. Eng. Knowl. Eng. 2006.16:523-552. Downloaded from www.worldscientific.com
by UNIVESITY OF TEXAS AT DALLAS on 01/30/14. For personal use only.

we use a queue S_i to keep track of nodes that need to be traversed in the ith loop. The pseudo-code for the first phase is given below.

```
i = 0; if (q∈ S0) then q.distance = 0; else q.distance = ∞; endif;
while (Si != empty) do
   for-each q∈ Si do q.num = q.num + 1;
      if (q.mark = 0) then q.mark = 1; p = q's parent;
         if ( (q!= root) && (d(p,q) + q.distance ≤ D) ) then
            if (d(p,q) + q.distance < p.distance) then p.distance = d(p,q) + q.distance; endif;
            put p in Si+1;
         endif;
      endif;
   endfor;
   i = i + 1;
endwhile;
if (q∈S0) then q.num = q.num − 1; endif;
```

Once all the nodes have been marked, we traverse T again starting from S_0. This time, nodes are clustered and the selected representative nodes are placed in S_f. During the traversal, a node is not processed till all its children nodes have been processed. Consider node q and its parent node p. If the traversal distance is going to exceed D, i.e., $q.distance \leq D$ and $p.distance > D$, then the subtree with q being the root may have to form a group (the group only includes nodes in S_0). If $q \in S_0$ and $q.sum \geq M$, where $q.sum$ is the total number of nodes in the subtree that are in S_0, then the group can be formed and we put q in S_f. Subsequently, p should no longer consider q as a marked child. In the case where $q \notin S_0$ or $q.sum < M$, a more complicated procedure is needed. The pseudo-code for the second phase is given in the following:

```
i = 0;
while (Si != empty) do
   for-each q ∈ Si do
      if (q.num = 0) then p = q's parent; p.num = p.num − 1; //one child has been computed.
         if ( (q = root) || (d(p,q) + q.distance > D) ) then // should not traverse up any further
            if (q∈S0) then if (q.sum ≥ M) then put q in Sf; else recollect (q, 1, i, 0); endif;
            else get_representative (q, i); endif;
         else if (d(p,q) + q.distance ≤ D) then p.sum = p.sum + q.sum; //traverse as normal
                  if (p∉ Si+1) then put p in Si+1; endif;
                  if (d(p,q) + q.distance > p.distance) then p.distance = d(p,q) + q.distance;
                  endif;
               endif;
            endif;
         else if ( (q.num > 1) && (q∉ Si+1) ) then put q in Si+1; endif;
         endif;
      endfor;
      i = i + 1;
   endwhile;
```

Int. J. Soft. Eng. Knowl. Eng. 2006.16:523-552. Downloaded from www.worldscientific.com
by UNIVESITY OF TEXAS AT DALLAS on 01/30/14. For personal use only.

If the subtree with q as the root is to form a group, but $q \notin S_0$, then it is not suitable to use q as the group representative. Function "get_representative" is then used to decide whether each of q's children nodes (only marked ones) can be the group representative of its subtree.

```
get_representative (q, i)
    for-each r = q's child do
        if ( (r.mark = 1) && (r ∈ S0) ) then
            if (r.sum ≥ M) then put r in Sf; else recollect (r); endif;
        else get_representative (r, i); endif;
    endfor;
```

Note that if q's subtree cannot form a group due to $q.sum < M$, then the nodes in the subtree that are in S_0 need to be recollected and placed in S_f (to be displayed individually as the search researchs). The same recollection is needed for node r in the function above. Function "recollect" simply traverses the tree and puts the corresponding nodes in S_f.

An example is given here to illustrate the grouping algorithm. Consider the subtree shown in Fig. 9. The nodes in S_0 are shaded. The distances between two connected nodes are marked on the corresponding edge, e.g., $d(p_3, p_5) = 1.5$, $d(p_2, p_3) = 0.3$, etc. Assume that D and M are set to 2 and 3, respectively.

The first phase of the grouping algorithm computes the number of children nodes for each node. In the second phase, the traversals start from the leaf nodes (with "number of children nodes" $= 0$) in S_0 and proceeds upward toward the root node. In the example subtree, the leaf nodes include p_3, p_6, p_9, p_{13} and p_{16}. Note that p_3 is a leaf node since its children are nonmatching nodes and its distance to

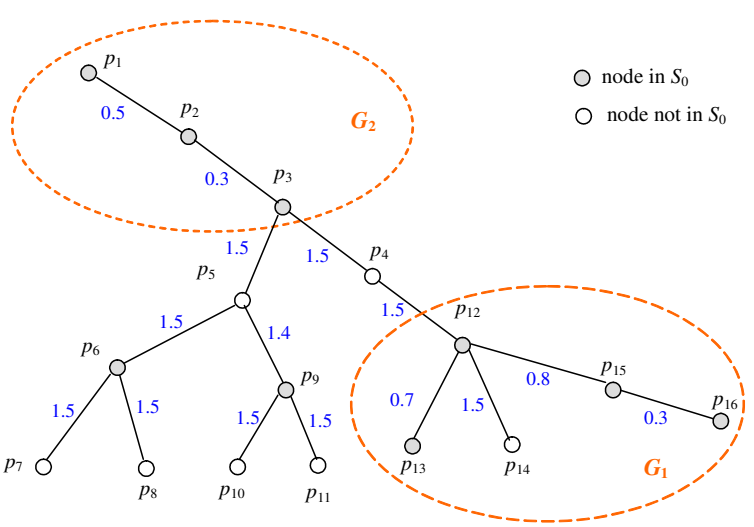

Fig. 9. An example of the search results grouping.

Int. J. Soft. Eng. Knowl. Eng. 2006.16:523-552. Downloaded from www.worldscientific.com
by UNIVESITY OF TEXAS AT DALLAS on 01/30/14. For personal use only.

Int. J. Soft. Eng. Knowl. Eng. 2006.16:523-552. Downloaded from www.worldscientific.com
by UNIVESITY OF TEXAS AT DALLAS on 01/30/14. For personal use only.

the nearest descendant matching node is greater than D. We first start from node p_{16}, traverse upward to p_{15}. Because the distance $d(p_{16}, p_{15}) = 0.3 < D$ and p_{15} has no other children nodes, the traversal continues to p_{12}. During the traversal, the traversal distance is accumulated and $d(p_{16}, p_{12}) = 0.3 + 0.8 = 1.1$. Similarly, we need to traverse from p_{13} to p_{12} and the traversal distance along this path is 0.7. Now node p_{12} has been marked as "traversed from all children nodes in S_0", it sets its traversal distance to the maximum distance of all paths, which is 1.1. Since p_{12}'s traversal distance is less than D, we continue to traverse up to its parent node p_4. However, $d(p_4, p_{12}) = 1.5$ which makes $d(p_4, p_{16}) = 2.6 > D$, thus, p_4 cannot be grouped with p_{12} (note that in this case, p_4 is not in S_0 and will be excluded even if the traversal continues). Now the traversal terminates and p_{12} becomes the leading node. Let G_1 denote the subtree including p_{12} and all its descendants in S_0. Since the distance between p_{12} and each node in G_1 is less than D, p_{12} is selected as the representative node of subtree G_1. Next we consider nodes P_6 and P_9. We traverse from p_6 to p_5 and from p_9 to p_5. Now node p_5 has been marked as "traversed from all children nodes in S_0", it sets its traversal distance to the maximum distance of all traversed paths, which is 1.5. Since p_5's traversal distance is less than D, we continue to go up to node p_3. But $d(p_5, p_3) = 1.5$ which makes $d(p_6, p_3) = 3.0 > D$. Thus, p_3 cannot be grouped with p_5 and the traversal terminates. But p_5 cannot be selected as the representative node because it is not in S_0. Hence, p_6 and p_9 are recollected. Finally, we start traversal from p_3 and go up to p_2 and then to p_1. Let G_2 denote the subtree including p_1, p_2 and p_3. Since the distance between p_1 and each node in G_2 is less than D, p_1 is selected to be the representative node of G_2. As it reaches the root of the tree, the grouping algorithm terminates with the final representative nodes in S_f which includes p_1, p_6, p_9, and p_{12}.

4. Component Selection Based on Nonfunctional Requirements

One major goal of ORES is to facilitate component composition. To compose a system, the developer has to consider both functional and nonfunctional requirements of the system. ORES browsing and search features facilitate the identification of components that satisfy the functional requirements. In addition, ORES also enables users to analyze the nonfunctional properties and to select components that best satisfy the nonfunctional requirements of the system [14].

In ORES, the user can prepare a composition specification, which defines how to compose the components to achieve a goal function. We use the synchronous data flow model (SDF) as discussed in Ptolemy [2] for the specification of embedded system composition. In Ptolemy, the components are represented by actors. The actors interact through I/O buffers. Each actor consumes a fixed number of tokens from its input ports before execution and writes a fixed number of tokens to its output ports after execution. In our system, actors are replaced by component templates. A component template is a virtual unit in the composition specification. It has to be instantiated by an actual component in the repository. The purpose of

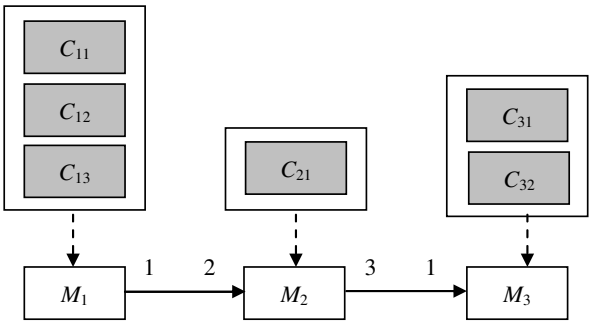

Fig. 10. An example composition specification in SDF model.

Int. J. Soft. Eng. Knowl. Eng. 2006.16:523-552. Downloaded from www.worldscientific.com
by UNIVESITY OF TEXAS AT DALLAS on 01/30/14. For personal use only.

component templates is to allow flexible selection of components for a given composition. For each component template, the user can search and select appropriate components in ORES and associate them with it. Then, the system can analyze different combinations of component selections and select the final set of components for instantiating the component templates. In Fig. 10, an example SDF specification is given. M_1, M_2, and M_3 are component templates. M_1 is associated with a list of components C_{11}, C_{12}, C_{13} that can realize the functionality defined by M_1. Similarly, M_2 is associated with component C_{21} and M_3 is associated with a list of components C_{31}, C_{32}. M_1 and M_2 are connected by a connector, which represents the I/O buffer in SDF. M_1 produces 1 token during each invocation and M_2 consumes 2 tokens during each invocation (i.e., M_2 can only be activated after two invocations of M_1). Similarly, M_2 and M_3 are connected and M_2 produces 3 tokens during each invocation and M_1 consumes 1 token during each invocation.

The nonfunctional properties of each component are needed for composition analysis to determine the final selections of components to instantiate the component templates. Each domain considers different nonfunctional properties. Thus, a specific DTD file (that extends the basic DTD defined in Fig. 7) is needed for each domain to specify the components and the measurement data. In Fig. 11, we present a DTD file for the components in the SDF model. The specification part of Fig. 7 is extended to include two elements "Name" and "IOBuffer". "Name" simply specifies the name of the component while "IOBuffer" specifies the input and output buffers and includes fields "Unit", "InputBuffer", and "OutputBuffer". The unit of buffer sizes is specified by the "Unit" tag. "InputBuffer" and "OutputBuffer" can appear as many times as necessary (including zero time). "InputBuffer" is extended to "Name" and "InputBufferSize". The "Name" tag specifies the name of the buffer which is uniquely defined within a component. "InputBufferSize" specifies the size of the input buffer in terms of the given unit (the same as the number of tokens). Similarly, "OutputBuffer" is extended to "Name" and "OutputBufferSize".

The properties part of Fig. 7 is extended to include typical nonfunctional properties of the components in the SDF model, including "Memory" and "Time". "Unit"

```
<!ELEMENT Specification (Name, IOBuffer)>
<!ELEMENT Name (#PCDATA)>
<!ELEMENT IOBuffer (Unit, InputBuffer *, OutputBuffer *)>
<!ELEMENT Unit (#PCDATA)>
<!ELEMENT InputBuffer (Name, InputBufferSize)>
<!ELEMENT Name (#PCDATA)>
<!ELEMENT InputBufferSize (#PCDATA)>
<!ELEMENT OutputBuffer (Name, OutputBufferSize)>
<!ELEMENT Name (#PCDATA)>
<!ELEMENT OutputBufferSize (#PCDATA)>
<!ELEMENT Properties (Memory, Time)>
<!ELEMENT Memory (Unit, Measurement)>
<!ELEMENT Unit (#PCDATA)>
<!ELEMENT Measurement (#PCDATA)>
<!ELEMENT Time (Unit, Measurement)>
<!ELEMENT Unit (#PCDATA)>
<!ELEMENT Measurement (#PCDATA)>
```

Fig. 11. The DTD file for components in the SDF model.

Int. J. Soft. Eng. Knowl. Eng. 2006.16:523-552. Downloaded from www.worldscientific.com
by UNIVESITY OF TEXAS AT DALLAS on 01/30/14. For personal use only.

and "Measurement" tags are used for each type of nonfunctional properties, where "Unit" specifies the measurement unit (e.g., number of bytes or CPU cycles) and "Measurement" specifies the actual measurement data.

An example XML component specification based on the DTD file given above is shown in Fig. 12. In this example, the component requires 100 bytes of memory (excluding the external I/O buffer requirements) and its execution time is 125 CPU cycles. It has one input buffer and one output buffer and the input buffer and output buffer sizes are 2*10 and 3*10 bytes, respectively.

Multiple components may be candidates for instantiating a component template and each of them has different nonfunctional properties. Thus, selecting different combinations of components can yield tradeoffs of nonfunctional properties of the overall system. Above ORES, a Composition Analyzer [14] is designed to analyze the impact of different component selections on the properties of the composite system and to determine the most appropriate component selections. The analyzer works as follow:

(1) The analyzer first determines a potential set of candidate components for instantiating the component templates (one component for each template) in the composition specification of the system. Consider the example system specification shown in Fig. 10. The initial configuration can be, for instance, C_{11} instantiates M_1, C_{21} instantiates M_2 (no other choices), and C_{32} instantiates M_3.
(2) Based on the selected components, the nonfunctional properties of the composed system are analyzed. The analyzer takes the XML specification files of

```
<Specification>
    <IOBuffer>
        <Unit>10 bytes</Unit>
        <InputBuffer>
            <Name>M1&M2</Name>
            <InputBufferSize>2</InputBufferSize>
        </InputBuffer>
        <OutputBuffer>
            <Name>M2&M3</Name>
            <OutputBufferSize>3</OutputBufferSize>
        </OutputBuffer>
    </IOBuffer>
</Specification>
<Properties>
    <Memory>
        <Unit>byte</Unit>
        <Measurement>100</Measurement>
    </Memory>
    <Time>
        <Unit >CPU cycle</Unit>
        <Measurement>125</Measurement>
    </Time>
</Properties>
```

Fig. 12. An example of nonfunctional properties specification of a component.

Int. J. Soft. Eng. Knowl. Eng. 2006.16:523-552. Downloaded from www.worldscientific.com
by UNIVESITY OF TEXAS AT DALLAS on 01/30/14. For personal use only.

the selected components as input. Different composition models require different analysis methods. For the SDF model considered here, the analyzer needs to first determine a schedule that can minimize the buffer usage [2]. Based on the schedule, the memory requirements and execution time can be computed [2].

(3) Different combinations of the components are selected as the candidates for composition analysis. We use genetic algorithm to choose promising candidates to be evaluated [15]. This is necessary when there are a large number of component templates and candidate components to be considered. Also, the components themselves may be configurable which requires additional parameters to be considered.

(4) The analysis results for the selected candidate component sets are compared against each other and the solution that best satisfies the system requirements is chosen.

5. ORES Implementation

We have implemented ORES in Java. It consists of a client site system and a server site system. The architecture of ORES is shown in Fig. 13.

The client site system includes the ORES Explorer and a Composition Specification Tool. The ORES Explorer is a Java Applet that interfaces with the user and

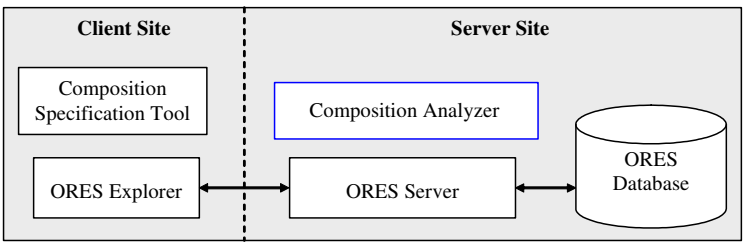

Fig. 13. The components in ORES.

Int. J. Soft. Eng. Knowl. Eng. 2006.16:523-552. Downloaded from www.worldscientific.com by UNIVESITY OF TEXAS AT DALLAS on 01/30/14. For personal use only.

communicates with the server to process user requests. The Composition Specification Tool provides a GUI interface for component composition specification.

The Server site system includes the ORES Server and a Database. The ORES Server provides the services of ontology browsing and navigation, component search, component viewing, etc. The ORES Database maintains the components and their relations. It also supports update operations such as adding, removing, and updating the ontology. The Composition Analyzer is also a part of the server site system. It works on top of the ORES Server as a plug-in tool. The details of the Composition Analyzer have been presented in Sec. 5.

A user can prepare a composition specification for a system using the Composition Specification Tool. The system specification consists of component templates and connectors (as discussed in Sec. 4). The user can also select appropriate components using the ORES Explorer and associate them with the component templates. A set of components instantiating the component templates can be determined using the Composition Analyzer, resulting in a system that best satisfies the non-functional requirements of the system. The designs of the major components of ORES are discussed in the following subsections.

5.1. *ORES explorer*

The ORES Explorer is a Java Applet which enables users to browse the ontology, search for components, and view the components. A snapshot of the ORES Explorer interface is given in Fig. 14.

For efficient ontology browsing and components viewing, a partial ontology and part of the components information is maintained in the ORES Explorer. If the browsing range is out of the locally maintained ontology or some component information is not stored locally, then the ORES Explorer sends a request to the ORES Server to obtain the additional information. To add a partial ontology to the repository, the user can specify the partial ontology using an XML file and the file is transferred to the server and processed there. To add a component, the user needs to supply the required component information and the parent node-id. Functions for deleting nodes and links in the ontology are also provided. Also, the user can submit a set of keywords to search for the desired components.

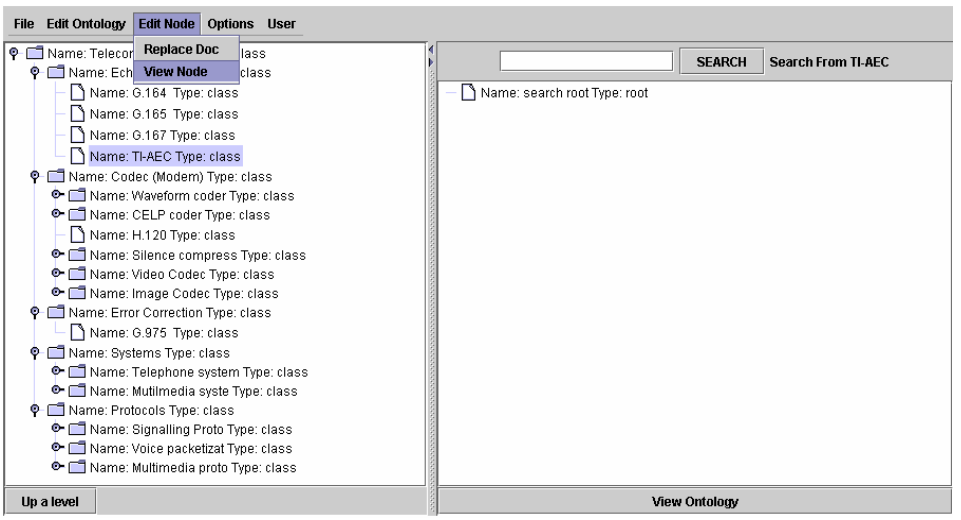

Fig. 14. The user interface of the ORES Explorer.

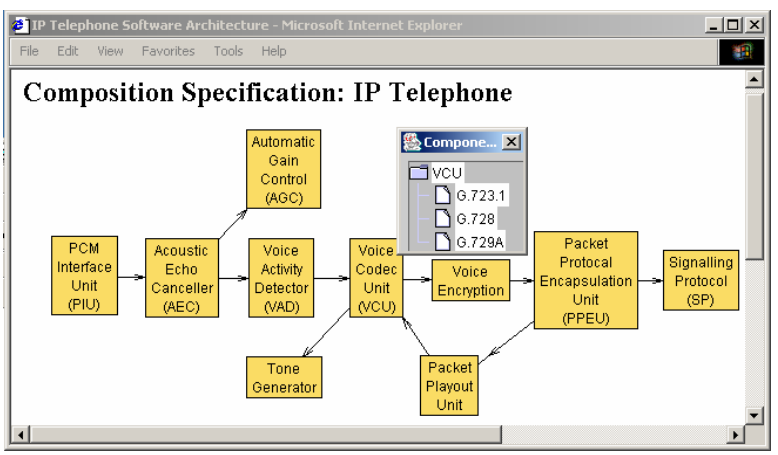

Fig. 15. User interface for the composition specification tool.

5.2. *Composition specification tool*

ORES provides a GUI interface for composition specification. Through the interface, a user can prepare a composition specification. The user first defines the required component template that composes the desired system and their I/O ports. Each component template can be associated with one or more components in the repository. The I/O ports of the components should match those of the templates. The I/O ports of the templates are connected through the GUI to form the final Synchronous Data Flow (SDF) specification of the system. Figure 15 shows

Int. J. Soft. Eng. Knowl. Eng. 2006.16:523-552. Downloaded from www.worldscientific.com
by UNIVESITY OF TEXAS AT DALLAS on 01/30/14. For personal use only.

the composition interface provided in ORES. A composition specification for an IP telephone subsystem is shown in the window. It consists of a set of component templates. As can be seen in the diagram, three components are associated with the component template "VCU" and any of them can be selected for system assembly. The Composition Analyzer can help determine the best selection of components for the templates [14].

Int. J. Soft. Eng. Knowl. Eng. 2006.16:523-552. Downloaded from www.worldscientific.com
by UNIVESITY OF TEXAS AT DALLAS on 01/30/14. For personal use only.

5.3. *ORES database*

The ORES Database is mainly responsible for maintaining the components and their relations in a persistent storage. An object-oriented database is frequently the choice for ontology storage. However, object-oriented databases are suitable for situations in which the behavior of the object is as important as the state of the object (attribute). In our case, we are concerned mainly with the state of the object. On the other hand, a relational database has the shortcomings that it supports only a flat file structure, and not a nested or hierarchical structure. Thus, it is necessary to know the full set of attributes during the design of the database schema. However, as we can see (from Sec. 2.1), some information fields in ORES nodes are node-specific and cannot be determined *a priori*. The alternative choice we made is to use an object-relational database to represent the ORES ontology. One of the features of an object-relational database system is that it can be used to support multi-valued attributes or a set of values for a particular attribute. Thus, we can map the node-specific information to a multi-valued attribute. The database system we use is Oracle8i, which is an object-relational database management system.

In order to map ontology nodes into the database, we create a table named *SampleTable*. It has the basic node attributes: NodeId, NodeType, Keywords, SpecificFeatures, ForwardLinks, BackwardLink, and Pointers. NodeId is the primary key. Each node is mapped as a tuple or a row in the table. Attributes Keywords, SpecifcFeatures, ForwardLinks, and Pointers are multi-valued attributes. ForwardLinks in a given node contain a set of its children's node ids. BackwardLink in a given node contains its immediate parent node-id. The SpecificFeatures attribute for a node possesses a set of self-descriptive domain dependent tags, along with values. Thus, not only the value, but also its tag, is stored as one of the values of SpecificFeatures, which circumvents the problem associated with a fixed set of attributes during the schema design.

The Database also supports update operations such as adding, removing, and updating the ORES ontology and the components. ORES is initialized with a basic ontology containing the root node and some well-known domains. A partial ontology can be inserted into the existing ontology using the ontology editing function. A user can specify a partial ontology (to be inserted) using an XML file. For removing a component or a subtree, the user can locate the corresponding node in the ontology by NodeId and then remove the component or the subtree under the component.

5.4. *ORES server*

The ORES Server supports ontology navigation and browsing, component search, component viewing, etc. To make these operations more efficient, the ORES Server also maintains the ontology in memory. The links that connect two nodes are stored in both vertex nodes in the ORES Database. When the server program starts, it reconstructs the repository ontology in memory and the links are represented by pointers. Node information that needs to be used for browsing the ontology or for search operations is loaded in memory to increase efficiency. Individual node information that is not used during traversal or search stays in the ORES Database to avoid overloading the memory. A hash table is maintained to keep track of all the keywords that define the components in the repository. For each keyword entry, a list of pointers is maintained to point to the components that are associated with the keyword. The hash table is also stored in the ORES Database and reconstructed when the server program starts.

When a user submits a set of keywords to search for the desired components, the ORES Server identifies these keywords from the hash table and selects the components associated with them. When the number of search results exceeds a threshold, the grouping algorithm presented in Sec. 3 is activated. A set of domain nodes will be presented to the user. The user, at this point, can choose to browse the ontology following the selected domain nodes or perform further search under a selected domain node. The system also allows the users to confine the search within a certain domain. The user can browse the ontology and select a node to start the search. In this case, the search will be limited to the ascendants of the node. From a list of search results, the user can select a node to retrieve its information if the node is the desired one or to initiate further exploration if the potential match may be the node's ascendants or descendants.

6. Related Work

Several web-based component repositories have been constructed [7, 16, 28] to facilitate access to reusable components. Some of these use a taxonomy to structure components to facilitate retrieval [7, 24, 28]. This taxonomy provides an effective way of locating generic components (domain-independent) that are well-known to programmers and corresponds well with their intuition. However, when a user is uncertain of the repository taxonomy, the simple keyword-based search provided in these systems is not sufficient. Approaches that use structural information to assist with the search have been proposed. In [24], taxonomic hierarchies are used to speed up keyword search. Faceted classification [23, 30] offers a different structuring mechanism. It defines attribute clauses that can be instantiated with different terms. Users search for components by specifying a term for each of the facets. The faceted approach divides the information space to make it easier to specify individual terms to represent components. But it is often hard for a user to find the right terms or the right combinations that will accurately describe the component to be retrieved.

Int. J. Soft. Eng. Knowl. Eng. 2006.16:523-552. Downloaded from www.worldscientific.com
by UNIVESITY OF TEXAS AT DALLAS on 01/30/14. For personal use only.

Int. J. Soft. Eng. Knowl. Eng. 2006.16:523-552. Downloaded from www.worldscientific.com
by UNIVESITY OF TEXAS AT DALLAS on 01/30/14. For personal use only.

In [33], we have proposed an ontology-based software component repository design. Ontology provides a powerful mechanism for specifying component relations to facilitate component retrieval. Other repository systems based on information systems techniques have also been proposed. In [32], Ye and Lo propose a retrieval mechanism composed of semi-automatic feature extraction, automatic indexing and self-organizing map (SOM). SOM is developed as a two-level neural network. Retrieval queries can be described in natural language. These descriptions are transformed to query vectors by automatic indexing. Each query vector is mapped onto a winning node on SOM. Software components attached to the winning node or the nodes adjacent to the winning node are selected as the candidates for the specified query. They have restricted browsing based on SOM; on the other hand, in our case, we use ontology which is multi-level and provides controlled vocabularies and guided search. Also, in ORES, we address issues in specifying nonfunctional properties.

Besides a centralized software repository, researchers have focused on software components distributed across multiple repositories. For example, Braga *et al.*, [3] have proposed an approach addressing the interoperability issues distributed across multiple repositories. They propose an integration layer based on the mediator technique to provide the binding of different components to their domain concepts. By exploiting ontologies, they address heterogeneities of various software components. The mediator approach can be used complementarily with ORES to offer multiple repository retrievals.

7. Summary and Future Research

Effective component retrieval technique is crucial to the success of component-based software development. We have developed an online repository for embedded software (ORES) to facilitate the retrieval of embedded software components. We use an ontology-based approach to organize the software components in the repository. Since the ontology approach can typically have many views, it may not be suitable for navigation and browsing. We analyze the characteristics of the ontology that is suitable for software components and develop an echoing scheme to restructure the ontology to build a hierarchy that allows browsing with clear semantics of components relations. The repository also provides effective search operation. To eliminate the problem of obtaining a large number of search results without reducing the recall factor, we have developed a search results categorization approach. Instead of pruning the search results, we group the results and only display the root node of a subtree in a hierarchy. The user can then direct the search by providing the paths that he or she desires to explore further. We have developed an algorithm for grouping results based on the ORES ontology.

Conventional component specifications generally focus on functional properties. For embedded software, we also need to focus on nonfunctional specifications for the components. We have developed an XML-based specification model for both

functional and nonfunctional specifications of a component. We have identified measurable nonfunctional attributes and the parameters that impact them. The specification essentially describes the measurements of the nonfunctional attributes with various settings of the parameters. To conserve storage space, we use functions to describe the relationships between the nonfunctional attributes and parameters. The system we have developed can effectively represent the measurement data.

Currently, the measurement data are provided by the users and ORES provides various methods for their storage. Measurement of non-functional properties itself is a major research area. Measurement of some non-functional properties, such as security and safety, have not yet been well developed. Also, tools are needed to collect measurement data for components that are in the repository. We are conducting research on the measurement of non-functional properties and developing tools above ORES for measurement data collection.

We are also conducting research and developing techniques and tools for automated component selection and configurable parameter setting. When there are multiple choices of components that can be used to implement a required function, we analyze the nonfunctional properties of the component and their effects on the overall system to make a selection decision. Similarly, we analyze the effect of parameter settings of the component on the nonfunctional properties of the system to determine the proper parameter selections. The selection decision process is also backed up by the nonfunctional property specification in ORES for the components in the repository.

Another research direction related to ORES is automated code synthesis. We use the Synchronous Data Flow (SDF) model to specify component composition. The composition is first analyzed to select the best memory allocation strategy and execution schedule. The system code can be automatically generated according to the execution schedule. We also consider automated code synthesis for an extended SDF model that considers events.

References

1. K. Bergner, A. Rausch, and M. Sihling, Componentware — The bit picture, *20th ICSE Workshop on Component-Based Software Engineering*, Japan, 1998.
2. S. S. Bhattacharya, P. K. Murthy, and E. A. Lee, Synthesis of embedded software from synchronous dataflow specifications, *J. VLSI Signal Processing* **21** (Kluwer Academic Publishers, 1999), pp. 155–166.
3. R. Braga, M. Mattoso, C. Werner, The use of mediation and ontology technologies for software component information retrieval, *ACM Symposium on Software Reuse*, May 2001.
4. A. W. Brown and B. Barn, Enterprise-scale CBD: Building complex computer systems from components, in *Proc. Software Technology and Engineering Practice*, September 1999, pp. 82–91.
5. M. Bunge, *Treatise on Basic Philosophy, Ontology I: The Furniture of the World*, vol. 3, Reidel, Boston, 1977.

Int. J. Soft. Eng. Knowl. Eng. 2006.16:523-552. Downloaded from www.worldscientific.com
by UNIVESITY OF TEXAS AT DALLAS on 01/30/14. For personal use only.

6. P. S. Chen, R. Hennicker, and M. Jarke, On the retrieval of reusable software components, in *Proc. 2nd Int. Workshop on Software Reusability*, Italy, March 1993, pp. 99–108.
7. Component Source, http://www.componentsource.com.
8. L. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos, *Non-Functional Requirements in Software Engineering*, Kluwer Academic Publishers, 2000.
9. W. Frakes and R. Baeza-Yates, *Information Retrieval: Data Structures and Algorithms*, Prentice Hall, 1992.
10. L. Khan, D. McLeod, and E. Hovy, Retrieval effectiveness of ontology-based model for information selection, to appear in *The International Journal on Very Large Databases*, Springer-Verlag.
11. E. A. Lee and D. G. Messerschmitt, Pipeline interleaved programmable DSP's: Synchronous dataflow programming, *IEEE Trans. Acoustics, Speech, Signal Processing*, **35**, September 1987.
12. Y. Li, Toward a qualitative search engine, *IEEE Internet Computing* **2**(4) (1998) 24–29.
13. Luqi and J. Guo, Toward automated retrieval for a software component repository, in *Proc. IEEE Conf. and Workshop on Engineering of Computer-Based Systems*, March 1999.
14. H. Ma, I.-L. Yen, F. Bastani, and K. Cooper, Composition analysis of QoS properties for adaptive integration of embedded software components, in *Proc. IEEE Int. Symp. on Software Reliability Eng.*, Denver, Colorado, November 2003.
15. H. Ma, D. Wang, F. Bastani, I.-L. Yen, and K. Cooper, A model and methodology for composition QoS analysis of embedded systems, in *11th IEEE Real Time and Embedded Technology and Applications Symposium* (*RTAS'05*), March 7–10, 2005, pp. 56–65.
16. Microsoft, Microsoft Repository, www.microsoft.com/repository.
17. A. Mili, R. Mili, and R. Mittermeir, Storing and retrieving software components: A refinement based system, in *Proc. Int. Conf. Software Engineering*, May 1994, pp. 91–100.
18. D. I. Moldovan and R. Mihalcea, Using WordNet and lexical operators to improve Internet searches, *IEEE Internet Computing* **4**(1) (2000) 34-43.
19. E. Ostertag, J. Hendler, R. Prieto-Diaz, and C. Braun, Computing similarity in a reuse library system: An AI-based approach, *ACM Transactions on Software Engineering and Methodology* **1**(3) (1992) 205–228.
20. K. Ozeka and T. Umeda, An adaptive filtering algorithm using an orthogonal projection to an affine subspace and its properties, *Electronics and Communications in Japan*, **67-A**(5) (1984).
21. L. Perrochon, A quick tutorial on searching and evaluating Internet resources, *IEEE Communications Magazine* **35**(6) (1997) 142–145.
22. A. Pretschner and S. Gauch, Ontology based personalized search, in *Proc. IEEE Int. Conf. on Tools with Artificial Intelligence*, 1999, pp. 391–398.
23. R. Prieto-Diaz and P. Freeman, Classifying software for reusability, *IEEE Software* **4**(1) (1987) 6–16.
24. W. Rossak and R. T. Mittermeir, A DBMS based repository for reusable software components, in *Proc. 2nd Int. Workshop on Software Engineering and its Applications*, France, 1989, pp. 501–518.
25. J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley, December 1998.
26. Smalltalk protocol, http://www.smalltalk.org

Int. J. Soft. Eng. Knowl. Eng. 2006.16:523-552. Downloaded from www.worldscientific.com
by UNIVESITY OF TEXAS AT DALLAS on 01/30/14. For personal use only.

27. R. A. Steigerwald, Reusable component retrieval for real-time applications, in *Proc. IEEE Workshop on Real-Time Applications*, May 1993, pp. 118–120.
28. Sun Microsystems, The source for Java technology, http://java.sun.com.
29. Q. Tran and L. Chung, NFR-Assistant: Tool support for achieving quality, *IEEE Symp. Application-Specific Systems and Software Engineering and Technology*, Texas, March 1999, pp. 284–289.
30. P. Vitharna, F. Zahedi, and H. Jain, Knowledge based repository scheme for storing and retrieving business components: A theoretical design and an empirical analysis, *IEEE Transactions on Software Engineering* **29**(7) (2003).
31. Y. Wang and M. Kitsuregawa, Use link-based clustering to improve web search results, in *Proc. Int. Conf. Web Information Systems Engineering* **1** (2001) 115–124.
32. H. Ye and B. W. N. Lo, Towards a self-structuring software library, in *IEEE Proc. on Software Engineering* **148**(2) (2001).
33. I.-L. Yen, L. Khan, B. Prabhakaran, F. B. Bastani, J. Linn, An on-line repository for embedded software, *IEEE Int. Conf. on Tools with Artificial Intelligence*, Dallas, Texas, November 2001.
34. M. Zaremski and J. M. Wing, Specification matching of software components, *ACM Trans. Software Engineering and Methodology* **6**(4) (1997) 333–369.

Int. J. Soft. Eng. Knowl. Eng. 2006.16:523-552. Downloaded from www.worldscientific.com
by UNIVESITY OF TEXAS AT DALLAS on 01/30/14. For personal use only.