An On-line Repository for Embedded Software

I-Ling Yen, Latifur Khan,
Balakrishnan Prabhakaran, Farokh B. Bastani
Dept. of Computer Science, Univ. of Texas at Dallas
{ilyen, lkhan, praba, bastani}@utdallas.edu

John Linn
Texas Instrument, Inc.
linn@ti.com

Abstract

The use of off-the-shelf components (COTS) can significantly reduce the time and cost of developing large-scale software systems. However, there are some difficult problems with the component-based approach. First, the developers have to be able to effectively retrieve components. This requires the developers to have an extensive knowledge of available components and how to retrieve them. After identifying the components, the developers also face a steep learning curve to master the use of these components. We are developing an On-line Repository for Embedded Software (ORES) to facilitate component management and retrieval.

In this paper, we address the issues of designing software repository systems to assist users in obtaining appropriate components and learning to understand and use the components efficiently. We use an ontology to construct an abstract view of the organization of the components in ORES. The ontology structure facilitates repository browsing and effective sea rch. We also develop a set of tools to assist with component comprehension, including a tutorial manager and a component explorer.

1 Introduction

Software technology is rapidly shifting away from low-level programming issues to automated code generation and integration of systems from components. The component-based approach can significantly reduce the cost and time for software development. However, it also has some difficult problems. First, the developers have to be able to effectively retrieve related components. The retrieval process involves matching the desired functionality and making sure that the component satisfies required properties such as timing and resource constraints. Thus, the developers need to have in-depth knowledge of the available components and their properties. After identifying the components, the developers also face a steep learning curve to master the use of them.

Effective component retrieval technique is crucial to the success of component-based approach. Over the past decade, component retrieval has been studied extensively [1,8]. Desirable retrieval techniques should yield high precision and recall [5]. 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 $||R \cap I|| / ||R||$, i.e., requires the retrieval algorithm to return only the relevant components. Recall can be defined as $||R \cap I|| / ||I||$, i.e., requires the retrieval algorithm not to miss any relevant components [5].

Formal methods have been used [7,8] 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 [5,8]. Since it does not provide a complete behavior description, it is not suitable for partially specified retrievals. Semantics-based approach specifies a component by its behavior. Generally, formal methods are used for behavior specification [1,8]. Theorem proving or rule-based reasoning techniques can be used to infer equivalence or similarity of component behaviors [9]. These are elegant solutions; however, they require the component developers and users to have extensive knowledge of formal specification techniques and are difficult to use due to the low-level granularity of formal specification.

Several web-based component repositories have been constructed [2,3,6,12] to facilitate access to reusable components. Some use a taxonomy to structure components to facilitate retrieval [2, 11, 12]. 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, 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 [11], taxonomic hierarchies are used to speed up keyword search. Faceted classification [10] 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. In enumerated classification, information is divided into categories, and then subcategories, to form a structured hierarchy.

In our approach, we use an ontology to organize the components in the repository. Ontology is a collection of nodes and their relationships, which collectively provide an abstract view of a certain application domain. It has a similar structure as that in enumerated classification and, thus, facilitates effective browsing. Also, we use ontology-based search, so that the search scheme fully exploits the meta-information offered in the ontology and improves the level of precision and recall.

Most component retrieval researches focus on functionality match. However, for some applications, such as embedded software systems, component retrieval generally involves nonfunctional requirements (NFR), such as time limit, memory constraints, security requirements, etc. Thus, the design of component repository and the retrieval techniques should also consider the satisfaction of NFR. Little work has been done along this direction. In [14], an NFR-Assistant tool is presented which assists with the exploration of design alternatives through a graphical interface. In [13], the real-time requirement is addressed and tools are provided for selection of components satisfying real-time constraint. However, both systems use exhaustive search to identify components which incurs intensive computation.

In this paper, we discuss the design and development of an On-line Repository for Embedded Software (ORES). Our design principles include:

(a) The system should support effective component retrieval. We consider retrieval by browsing and by searching. We use an ontology-based repository structure [4] to capture various types of relations in software components and, subsequently, to facilitate effective component retrieval.
(b) The system should facilitate component retrieval for satisfying a set of nonfunctional requirements. Nonfunctional attributes for each component should be captured in the repository and mechanisms for efficient component selection should be provided.
(c) To allow easy understanding and use of the components, the repository also maintains tutorials for the components. The tutorial can be assembled flexibly upon request to satisfy different user needs, such as different levels of abstraction. Tools that facilitate the exploration of the components will also be provided.

The rest of the paper is organized as follows: In the next section, we discuss the major design concepts for ORES, focusing on the ontology design that facilitate browsing and search. In Section 3, the ontology-based search scheme is discussed. In Section 4, we discuss two sets of tools associated with the ontology for component comprehension. Section 5 states the conclusion of the paper.

2 Repository Ontology

2.1 Repository Ontology and Browsing

Repository ontology is crucial for effective component retrieval. However, existing techniques may not be directly applicable for software repository systems. First, software components hold additional relations beyond semantic nodes, which is due to the boundary of packages. For example, in a voice over IP system, the sender needs to *encode* the voice stream and the receiver needs to *decode* the stream. Nodeual-based ontology may capture the relation between different versions of "encoder" or "decoder" functions, but not the "*syntactical*" correlation between a specific pair of "encoder" and "decoder" functions from the same package that have to be invoked in pairs. A consequence of this issue is the necessity of providing multiple views in the ontology, for example, one hierarchy is based on the classification of functionality of components and another retains the boundaries of software packages. However, due to the complex structure, an ontology with multiple hierarchies is not suitable for browsing. In our approach, we use an **echoing** technique to merge the multiple views into a single hierarchy during browsing while still retaining the characteristics of multiple views.

We first consider a major repository hierarchy that retains software package boundaries. Each node in the repository ontology has a **type**, which can be domain, subdomain, package, abstraction group, abstraction, function group, or function. At the highest level, we have a repository root node. In the hierarchy, software packages are considered as the intermediate units. Under the root, we build the repository ontology by classifying packages according to their functions and application domains. This process is similar to conventional ontology construction. At this level, a node in the ontology represents a **domain** or a **subdomain**. A domain is a major software area, for example, embedded system, operating systems, etc. A subdomain further divides a domain into finer categories. Within a subdomain (or even a domain), there are various software **packages**. In general, a software package consists of a number of **abstractions**, where each abstraction is a program unit that encapsulates certain abstract concepts or behaviors together with some state information that is accessible within the abstraction. A group of **functions** is implemented in an abstraction to access the encapsulated state information and/or achieve certain goals of the abstraction. In a large package, there may be hundreds of abstractions. We use **abstraction groups** to structure abstractions in a package into a hierarchy. Similarly, a large abstraction can have a large number of functions. To allow easy retrieval of the functions, we use **function groups** to structure functions in an abstraction.

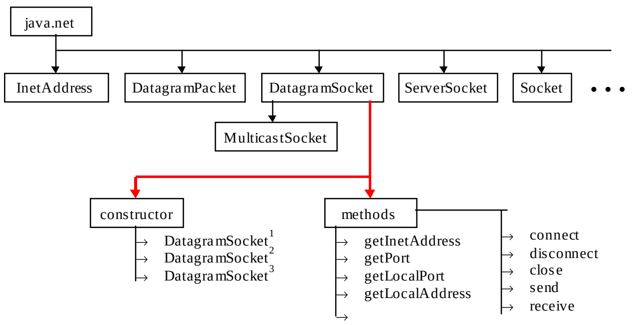


Figure 1. The hierarchy in package java.net.

Here, we use Java Networking package as an example to illustrate the ontology construction and discuss the echoing scheme in ORES. In Figure 1, we show the original program hierarchy in **java.net**. 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 illustrated. In ORES, the classes in java.net are abstractions. We further classify these abstractions into abstraction groups. Here, a sensible way to classify the classes is to have "reliable-communication", "unreliable-communication", and "multicast-communication". Within each class group, we further identify two sub class groups, "packet" and "socket". Under "unreliable-communication.packet", we have the actual class "DatagramPacket" and under "unreliable-communication.socket", we have the actual class "DatagramSocket". Within each class, we group functions into function groups. In the class DatagramSocket, we have function groups "constructors", "get/set-socket-address-information", "channel-establishment-functions", and "message-passing-functions". The ontology in ORES for the java.net package is shown in Figure 2.

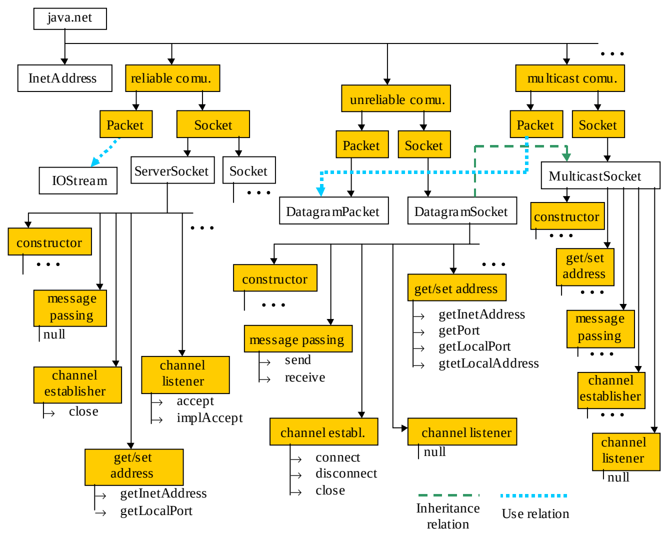


Figure 2. The ontology of Java.net package in ORES.

Another issue in software repository ontology is that software components may have their own hierarchy due to inclusion or inheritance. In Figure 2, we can see the inheritance and use relations. In a large repository, a subsystem consisting of components from the repository can also be a component in the repository. In conventional information hierarchy, the actual object can only be a leaf node. In software component hierarchy, components can be associated with nodes at any level in the ontology.

In Figure 2, nodes that belong to echoed hierarchies are shaded. The **echoing** scheme is used to build the same ontology for a group of nodes with similar behavior. To avoid information loss, echoed ontologies are constructed by taking the "union" of the individual ontologies being echoed. In java.net, the abstractions *ServerSocket*, *DatagramSocket*, and *MulticastSocket* have similar concepts. If we consider component semantics instead of package boundary, then a different hierarchy will be constructed. This hierarchy forms a second view of the ontology. The ontology with multiple hierarchies is presented in Figure 3. When multiple hierarchies of the ontology are displayed in a single view, browsing the ontology can be very confusing. Thus, we define a single hierarchy that includes all nodes in the three abstractions. As shown in Figure 2, we add message-passing group in ServerSocket and message-listener group in DatagrameSocket and MulticastSocket abstractions. The added nodes will be empty and their purpose is to relate similar units in a uniform way. So, all socket classes have the same ontology and the nodes in these echoed ontologies are correlated. A pointer "echoed-node" is used to link the root nodes of echoed ontologies together.

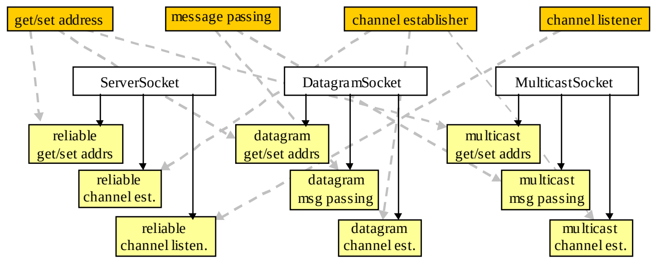


Figure 3. Multiple views in the ontology for Java.net

As we can see, ORES ontology provides a better structure for browsing. With the categorization within the packages and classes, relations among components are better captured. With the echo technique, users can easily understand the relations among various categories and compare them conveniently. Also, once users understand one group of components, it is easy to extend the knowledge to other echoed groups.

2.2 Components Description

Many information attributes are required to describe a component and different components may require different attributes to explicitly describe their characteristics. Thus, we use a flexible approach to dynamically define the attributes for each component. In ORES, DTD (document type definition) is used for information attributes definition. At the repository root node, we define a basic set of information attributes that 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. The DTD for the repository root node is as follows.

```
<!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 Properties (#PCDATA)>
<!ELEMENT Specification (#PCDATA)>
  . . . . . .
```

The information attributes include four categories, the general information, information pointers, node specification, and properties. In "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 has been defined in the previous subsection. Keywords field is maintained to facilitate search. In thecategory "pointers", a set of pointers is defined to link to external information files, such as code, tutorial, documentation for the component.

Specification attributes describe the characteristics of a node and they are node-type dependent. For function nodes, the API specification (input, output, exceptions) should be provided. For abstraction nodes, the abstraction API, such as interface functions and exceptions, and inheritable or externally accessible variables can be provided. For a package node, the package version number, new release information, licensing information, price information, vendor, etc. can be provided. The execution environment requirements, such as OS, processor speed, memory and disk size, devices, etc., should also be specified. Some package-wide glossary definitions can also be provided.

The property field for each node is domain-dependent. We consider some "non-functional properties" for embedded software components, including timing information, memory footprint, etc. Users can search for components that satisfy a given timing constraint and/or memory size limitation, etc. Reliability related information, such as component reliability, operational profile, test data, test coverage, etc., can be included. Informal review information, such as problem and bug reports, user reviews, product limitations, usage experience report, etc. can be given. We are currently designing the tools to facilitate automated collection and easy entry of component property data.

3 Repository Search

A user may browse the ontology to locate a node, but frequently, the user may not have extensive knowledge of the ontology and has to make use of the search operation. Each node in ORES is described by a concept, which is further defined by a set of keywords. These keywords are used for search. The keywords for each component are obtained by analyzing the component documents and source code. The frequency of keywords appearing in these documents is used to assign weights for the keywords. Ontology hierarchy relations are used to further direct the keyword weight assignment. A high-weight keyword for a node may be propagated to its parent and children nodes. An inverted list is maintained in memory to keep track of the keywords and their relations to the components to allow efficient search.

A problem with search is the potential of having a large number of search results. Various methods have been developed to prune search results. For example, we can prune search results based on a certain ranking mechanism. The nodes with ranks lower than a threshold are removed. This approach has the potential of reducing the recall level. Instead, we use a **search result clustering** approach to direct the user to go along the correct direction to get the desired search results. For example, consider searching for the term "agent". A huge number of matches in very diverse areas may come up in a general knowledge base. There may be "agent" in real-estate, in chemistry, and in computer science. Within computer science, there may be many different types of agents, such as AI, E-commerce, or networking domains. Instead of pruning the results automatically, our approach clusters the search results, identifies and presents the domain nodes that represent a group of search results, and lets the user make the choices for further exploration. This approach can lead the user to prune the irrelevant search results in a much more effective way and yields higher recall level.

Consider the repository as a tree (the main structure of the ontology is defined as a tree) and let T denote this tree. Also, let S_0 denote the set of search results and N the number of search results, where $N = |S_0|$. Also, let S_f denote the set of nodes that are selected to represent their descendants in S_0 and to be presented to the users for choices for further exploration. We first choose a bound B_N. If $N \leq B_N$, then the number of search results is reasonably small and clustering is not needed. B_N can be determined by the user. A reasonable choice can be a number between 30 and 50. When $N > B_N$, we start the clustering computation and the algorithm for computing S_f is discussed below.

The algorithm consists of two phases. In the first phase, we start from nodes in S_0, traverse up R, and mark the traversed nodes. We bound the levels to traverse up in R to L levels. Let $T(p, l)$ denote the set of nodes that are in in S_0 and within the l-level subtree of p and p itself. Also, let $N(p, l) = |T(p, l)|$. In the second phase, we compute $N(p, l)$ for some $l < L$ and for all p, where p is marked. Note that the $N(p, l)$ value for node p is not computed and not propagated up to its parent till all $N(q, l_q)$ from all q, where q is the child nodes of p, have been computed. The users can choose the bounds M_1, which is the minimum number of nodes that can form a group, and M_2, which is the maximum number of nodes that should form a group. The clustering decisions are made based on the chosen M_1 and M_2 values. M_1 and M_2 should be properly selected to achieve the best effect. If M_2 is too small, then there may be too many representative nodes to be presented. If M_2 is too large, then the user may have to go through too many screens before reaching the desired nodes. If only one bound M_1 or M_2 is used, the results may not be desirable either. If we only use M_1, then in a dense area, there may be too many groups. If we only use M_2, then in a sparse area it is hard to form groups. A reasonable way to view clustered groups is to present 15-30 groups at a time (depending on the value N). Let g denote the number of groups to be presented. Each group is thus of size N/g. So, a sensible choice for M_1 and M_2 can be $N/2g$ and $2N/g$, respectively. It is also important to choose a proper L value. If L is too large, then a group may include nodes that are far apart from its representative node and not being properly represented. If L is too small, then it is either hard to form groups or the group size is too small. Some more scenarios about the use of M_1 and M_2 are analyzed.

1. When $N(p, l) \geq M_2$, then even if $l < L$, p should be selected to represent its descendants. Consider the case in Figure 4(a). For nodes p_1, p_2, and p_3, we have $N(p_1, l) = 24$, $N(p_2, l) = 11$, and $N(p_3, l) = 12$. Assume that $M_1 = 3$,

M_2 = 10, and L = 4. As we can see, p_1 can be selected to represent the matched nodes in the entire subtree, but there are too many nodes in the subtree, instead, p_2 and p_3 are included in S_f, where $N(p_2, l)$ and $N(p_3, l)$ are greater than M_2. Node p_1, in this case, is considered with its ancestors to form another potential group.

2. Figure 4(b) presents a case that the nodes in S_0 are spread sparsely over a subtree rooted at node p_5. After the computation, node p_6 will have $N(p_6, 9) > M_2$; however, p_6 is too far from some of its descendant it represents, such as p_8. Thus, the computation should end at a bounded level, which is L. Assume that $L = 5$ and $M_2 = 5$. Therefore, nodes p_5 with $N(p_5, l) = 6 > M_1$ and p_7 with $N(p_7, l) = 6 > M_1$ will be included in S_f.

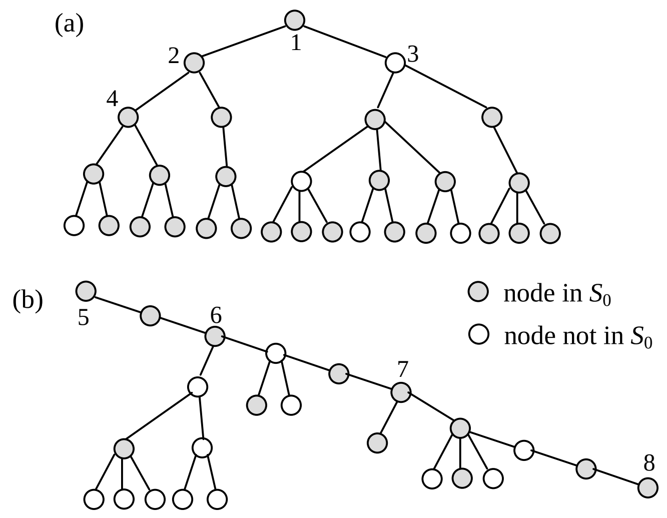


Figure 4. Two case.

The algorithm for computing S_f is given as follows. In the first phase, we traverse up R for L levels from the nodes in S_0 and mark all the nodes traversed (by setting *q.mark* to 1). For each node q, where q is marked, we compute *q.num* that is the number of q's immediate child nodes that are marked plus 1 (for q itself). Also, we use a queue S_i to keep track of nodes that need to be traversed at the i-th level.

```
loop $i = 0$ to $i = L-1$ do
  for-each $q \in S_i$ do
    set $q.num$ to $q.num + 1$;
    if ($q.mark = 0$) then $q.mark = 1$;  $p = q$'s parent;  put $p$ in $S_{i+1}$; endif;
  endfor;
endloop;
for-each $q \in S_L$ do set $q.num$ to $q.num + 1$; $q.mark = 1$; endfor;
```

Once all the nodes are marked, we traverse R again starting from S_0. This time, nodes are clustered and the selected representative nodes are placed in S_f. During traversal, a node is not processed till all its child nodes are processed. Note that *q.sum* is $N(q, l)$. When node q is processed, we check whether it should be selected as the group representative. If $N(q, l) > M_2$, for some $l < L$ or $N(q, L) > M_1$, then we put q in S_f and q's parent p will not consider q as a marked child. Otherwise, the traversal proceeds upward.

If the traversal exceeds L levels and comes to a node q, while $N(q, L) < M_1$, then we execute the *recollect* function to recompute $N(q, l)$ with a small l value ($< L$) in the attempt to cluster some nodes in the subtree of q with q's ancestors. Various algorithms can be used for recollection and the value l for recollection should be determined to minimize the potential of repetitive recollection.

```
loop $i$, from $i = 0$, till $S_i$ = empty do
  for-each $q \in S_i$ do
    set $q.num$ to $q.num - 1$;
    if ($q.num = 0$) then  // if $q.num > 0$, $q$ should wait
      if ($q.sum \ge M_2$) then
        put $q$ in $S_f$;  $p = q$'s parent;  put p in $S_{i+1}$;
      else if ($q.level \ge L$)  // should not traverse up any further
        if ($q.sum > M_1$) then put $q$ in $S_f$;
        else if ($q \in S_0$) then
          $sum = 1$;  $level = 1$;  $recollect(q, sum, level)$;
          $p = q$'s parent;  $p.sum = p.sum + sum$;  put $p \in S_{i+1}$;
          if ($level + 1 > p.level$) then $p.level = level + 1$;  endif;
        else discard ($q$);
        endif;
      else  //traverse up as normal case
        $p = q$'s parent;  $p.sum = p.sum + q.sum$;  put $p \in S_{i+1}$;
        if ($q.level + 1 > p.level$) then $p.level = q.level + 1$;  endif;
      endif;
    endif;
  endfor;
endloop;
```

4 Component Exploration

After identifying the components to be used, learning how to use them and how to integrate them is still a major challenge. Also, most components provided in large-scale libraries contain a rich collection of methods with many possible and complicated calling sequences. They provide generality and flexibility for various applications, but at the cost of being hard to use. Thus, tools that help with the comprehension of components should be provided. Here, we consider two sets of tools that facilitate component exploration: the tutorial manger and the component explorer.

4.1 Tutorial for a Component

4.1.1 Multimedia Tutorial Specification

ORES supports multimedia tutorial construction for the components to illustrate how each component works or how it should be used. We use an XML (eXtensible Markup Language) control file to specify the multimedia objects that need to be delivered as part of the tutorial as well as the timing relationships among the media objects that need to be maintained for achieving visual and aural synchronization during presentation. We use HTML+TIME to specify the temporal relationships for the tutorial presentation. HTML+TIME extends SMIL (Synchronized Multimedia Integration Language) where a set of extensions is added to allow additional timing, interaction, and media delivery capabilities. Using the timing extensions, any HTML element can be set to appear at a given time, to last for a specified duration, and to repeat (i.e. loop). Media tags are also introduced in SMIL to easily integrate time-based media (movies, audio and animation content). SMIL also introduces some very powerful elements that support conditional delivery of content, specifically to support differing client platform multimedia capabilities and preference settings. For added support for integration of timing and synchronization markup with other languages, we will use the Timesheets approach. Timesheets is a new concept under development. It separates timing from the content document's structure and provides a solution where time can be brought to any XML document regardless of its syntax and semantics.

4.1.2 Automatic Tutorial Assembly

Tutorial presentation may either be a description of the functionality of the selected component generically or a collective description of the sub-components that compose the selected component. For instance, consider the ontology given in Section 2.1. The tutorial for *DatagramSocket* may be composed of tutorials of its subcomponents *constructor*, *channel listener*, *channel establisher*, and *message passing functions*. Each of the subcomponent tutorial may in-turn be composed of the tutorials for each of the methods in the category. In ORES, we provide a mechanism for automatic tutorial assembly based on ontology. A user can use a variant XML file to specify a tutorial template. This template can be reused by nodes in the echoed ontology by replacing the variables defined in the file. For example, to assemble a tutorial for node DatagrameSocket, we can construct the following XML file:

```
<variables>class-name, constructor-file, listener-file, … </varialbes>
<content>
  The class #class-name, consists of functions for socket construction,
  channel listening, connection establishment, and message passing.
</content>
<htmlFile>#constructor-file.html</htmlFile>
<playFile>#constructor-file.wav</playFile>
<include>child 1</include>
<htmlFile>#listener-file.html</htmlFile>
<playFile>#listener-file.wav</playFile>
<include>child 2</include>
……
```

Each node using this template should define the variables. The variables are replaced by their defined values when the XML template is accessed. The <include> tag triggers an operation that fetches the tutorial of the designated child node. This template can be used by all socket classes such as *Socket*, *ServerSocket*, *DatagramSocket*, and *MulticastSocket* for dynamic tutorial assembly.

4.1.3 Tutorial Delivery

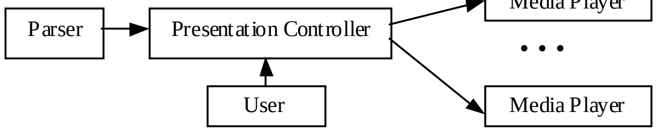


Figure 5. Modules in TutPre.

Tutorial presentation is carried out by the **TutPre** module. TutPre is developed using Java and the media objects comprising a tutorial are delivered using Java Media Framework (JMF). TutPre parses the XML control file of a tutorial and identifies the different media objects that are to be delivered for that tutorial. These media objects are then retrieved from the server and presented to the user, maintaining the timing relationships specified in the XML control file. TutPre has the structure shown in Figure 5. The parser module reads the XML control file and identifies the media objects and their timing relationships. This information (media objects and their timing relationships) is passed onto the *Presentation Controller*. The Presentation Controller creates an instance of a media player for each media type and synchronizes the delivery of media objects. It also interacts with the user to help him/her to skip, fast forward, or rewind during a tutorial presentation.

4.2 Component Explorer

Often, it is most effective to learn the characteristics and behavior of a component by hands-on exploration. In ORES, we develop tools to assist users who do not have much knowledge of the component to do effective exploration. To maximize the effectiveness, we first divide the component space into categories that are meaningful to users and allow them to explore each type of components in a specific way. The exploration tools need information regarding the behavior of the components. To allow the specification of the behaviors at a higher level of abstraction, we associate additional attributes with each component. These attributes denote high-level behaviors that are well defined and are incorporated into the tools that operate on the repository. The attributes are defined by partitioning the repository components into **abstract data type**, **stream**, **functional**, and **system** components. Components in the abstract data type (ADT) category encapsulate some state information and have methods that modify the state and other methods that return the value of some aspect of the state. ADTs can be specified using algebraic equations and can be further divided into **terminal**, **container**, **mapping**, and other categories as needed. Similarly, streams operate on a series of inputs that could be either signal (audio, video, etc.) or other information and perform filtering operations such as compression, encryption, routing, etc. Streams can be further divided into **data**, **audio**, **video**, and other groups, as well as tertiary attributes that specify the bandwidth and real-time response requirements of the streams. Functional components do not have any state, so the output depends only on the values of the input parameters. Examples include mathematical functions, sorting, searching, etc. System components are generic applications such as operating systems, databases, compilers, editors, etc., that are used in developing domain-specific applications. Based on this categorization, we will develop specific tools that will allow independent invocation of a component or a set of components to allow component comprehension, testing, or simulation. These tools can also be used to assess the component and obtain some of the property information (*P*) for the component.

Abstract data type object. The tool provides a capability for the user to specify the values of some aspect of the state of an ADT object instance and then automatically bring the object to the specified state. When a method is invoked on the object, the system will construct the input values (by querying the user, searching through the object space, random generation, etc.), invoke the operation, display the outputs and exceptions, and highlight all other methods that have been impacted. It also allows undoing a method invocation. These features can be used to facilitate component behavior observation, testing, stress evaluation, etc.

Stream object. The tool provides graphical interface to allow the connection of sources and sinks across one or more nodes. It also generates instructions to create, sustain, and delete streams. Based on the properties of the stream, it will generate or obtain values for the stream and enforce timing and reliability requirements.

Function object. The tool generates or obtains values for the inputs and creates new object instances to hold the result. Goal-oriented input data generator will also be provided for automated data generation.

5 ORES Implementation

5.1 Repository Implementation

The server site maintains the ontology in memory as well as in a database. When the server program starts, it reconstructs the repository ontology in memory from the database. 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.

For persistent storage, an object-oriented database system is frequently the choice for ontology storage. However, the object-oriented database is suited 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 Section 2.2), 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 multi-valued attributes. 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 for a node contain a set of children node ids. BackwardLink for a node contains 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.

5.2 User Functions

A user can browse through or edit the ontology, view or edit individual nodes, and search for components. For ontology browsing or components viewing, a partial ontology and part of the components information is maintained at the client site for efficient accesses. If the browsing range is out of the locally maintained ontology or some component information is not stored locally, then the client applet sends a request to the server to get 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 to be processed. 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.

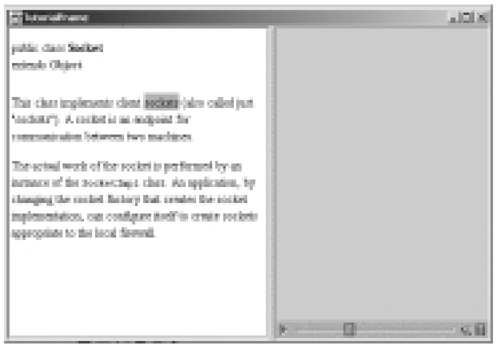
5.3 Search and Browsing

A user can submit a set of keywords to search for the desired components. These keywords are identified from the hash table and the components associated with them are selected. When the number of search results exceeds a threshold, the clustering algorithm discussed in Section 3 will be 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 in 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, a user can select a node from the search results to retrieve its node information if the node is the desired one or to start exploration if the potential match may be the node's ascendants or descendants.

5.4 Component Comprehension

We implemented the tutorial presentation system that provides multimedia presentation using Java JMF. For each component, an XML control file specifying the tutorial presentation media objects and synchronization information is provided. When user accesses a component and submits a tutorial presentation request, the presentation is activated. A presentation screen shots is given in Figure 6.


public class Socket
extends Object

This class implements client sockets (also called just "sockets"). A socket is an endpoint for communication between two machines.

The actual work of the socket is performed by an instance of the SocketImpl class. An application, by changing the socket factory that creates the socket implementation, can configure itself to create sockets appropriate to the local firewall.

Figure 6. Sample tutorial presentation window.

A prototype class tester for Java has also been developed. It uses introspection and reflection to determine the constructors and methods in the class and to facilitate invocation of the methods and monitoring of the outputs. It has been used successfully to evaluate existing and new Java classes that are ADTs. The class tester can be used effectively for exploration, such as component testing, understanding specific behavior of the component, and collect timing information for the components.

6 Conclusion

We have discussed the design of a software repository system ORES. Ontology is the framework for ORES that keeps all components in an organized structure to facilitate browsing and search. ORES also provides tools to facilitate tutorial generation and component exploration. Our contributions include:

1. We developed a systematic approach for ontology construction in ORES. The echoing scheme is used to retain the software components' boundaries and facilitates concepts categorization and association. Based on the ontology, it is easy to browse the repository to locate desired components.
2. We developed a search engine to locate components via ontology-based keyword search. We also developed an ontology-based search results clustering scheme to allow user-navigated, more effective search.
3. We developed the tools to assist component comprehension, including the tutorial manager and the component explorer. In the tutorial manager, we also introduce an automated tutorial assembly scheme to automatically compose tutorial of a parent node from the tutorials of its ascendants.
4. We implemented a prototype repository system. Currently, it uses basic algorithms to achieve ontology construction, search, tutorial management, and component exploration. In the future, a complete system will be implemented based on the concepts discussed in the paper.

7 Acknowledgement

This research was supported in part by the National Science Foundation under Grant No. CCR-9900922, by the Texas ATP under Grant No. 009741-0143-1999, and by Alcatel USA and Texas Instruments, Inc.

Bibliography

[1] P.S. Chen, R. Hennicker, and M. Jarke, "On the retrieval of reusable software components," *Proc. 2nd Intl. Workshop on Software Reusability*, Italy, March 1993, pp. 99-108.

[2] Component Source, http://www.componentsource.com/CS/Default.asp.

[3] EVB Software Engineering, "Ruse Library Toolset,", http://www.metronet.com/1/newprod/by-vendor/E/evb_software_e/

[4] L. Khan, "Structuring and Querying Personalized Audio using Ontologies," *Proc. of ACM Multimedia*, Volume 2, Orlando, FL, Nov 1999.

[5] Luqi and J. Guo, "Toward automated retrieval for a software component respository," *Proc. IEEE Conf. And Workshop on Engineering of Computer-Based Systems*, March 1999.

[6] Microsoft, "Microsoft Repository", www.microsoft.com/repository.

[7] A. Mili, R. Mili, and R. Mittermeir, "Sotring and retrieving software components: A refinement based system," *Proc. Intl. Conf. Software Engineering*, May 1994, pp. 91-100.

[8] Moormann-Zaremski and J.M. Wing, "Specification matching of software components," *ACM Trans. Software Engineering and Methodology*, Vol. 6, No. 4, 1997, pp. 333-369.

[9] 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*, vol. 1, no. 3, July 1992, pp. 205-228.

[10] R. Prieto-Diaz and P. Freeman, "Classifying Software for Reusability," *IEEE Software*, vol. 4, no. 1, pp.6-16, 1987.

[11] W. Rossak and R.T. Mittermeir, "A DBMS based repository for reusable software components," *Proc. 2nd Intl. Workshop on Software Engineering and its Applications*, France, 1989, pp. 501-518.

[12] Sun Microsystems, "The source for Java technology," http://java.sun.com.

[13] R.A. Steigerwald, "Reusable component retrieval for real-time applications," *Proc. IEEE Workshop on Real-Time Applications*, May 1993, pp. 118-120.

[14] 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.