JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 17, 147-158 (2000)

Short Paper

A Fully Distributed Approach to Repositories of Reusable Software Components

YUEN-CHANG SUN, MING-LIN KAO AND CHIN-LAUNG LEI
Department of Electrical Engineering
National Taiwan University
Taipei, Taiwan 116, R.O.C.
E-mail:{sun; kml; lei}@fractal.ee.ntu.edu.tw

Software reuse has long been a promising yet elusive technology for improving software productivity and quality. One of the related problems is the lack of quality software components and the inability of developers to efficiently find them. We attack this problem by establishing a fully distributed repositories architecture, with which component sharing and circulation are encouraged. Tools are developed to facilitate access to repositories across network, and techniques using degradation functions and a component migration history are introduced to further simplify component retrieval.

Keywords: component repositories, distributed systems, degradation functions, faceted classification, migration history, software reuse, thesaurus

1. INTRODUCTION

Software reuse involves the construction of new software systems using existing software artifacts. Both the products of previous software projects and the processes deployed to produce them can be reused; hence, a wide spectrum of reuse approaches exists [1, 2]. These approaches are customarily separated into two fundamental categories: *generative* and *compositional*. Our research is focused on the latter category.

The compositional methods involve reusable software components that are integrated into the target system. This type of software reuse is conceptually simple and straightforward. However, while reusable components have been shown to be beneficial in certain narrow areas, in general there has been less success [1]. To reuse software effectively, a large collection of reusable software is a necessity. Until recently, most research efforts have been focused on the development of a general-purpose "mega-library" that contains a large number of components, covers all application domains, is managed and used by a single organization, and does not communicate with other mega-libraries. Building such a large component repository, however, has its difficulties. First, the cost of building and managing a large general-purpose repository is

Received January 19, 1998; revised June 26, 1998; accepted August 11, 1998.
Communicated by K. S. Kuo.

so high that organizations are unwilling to invest in the effort. Second, finding the desired components among a large collection is so time-consuming that programmers are unwilling to try. In addition, the mega-library approach is not able to exploit the full potential of software reuse: while a component will not be re-invented in one organization, it can be invented as many times as there are organizations.

Recently the great success of the Internet has attracted the attention of many researchers in the reuse field [4-7]. Decentralized architectures have been introduced to connect repositories with the Internet so that they can share components, thus solving the above problems to some extent. This approach has several advantages. First, because a much larger number of components than one repository can offer is available to developers, it is more likely that an adequate component can be found. Second, the more often a component is used, the more hidden defects in it can be found and fixed [3]. By allowing users other than the owner to access it, a component can be used more frequently. Third, since to satisfy one requirement there can be more than one competing choice, less-reusable components will be less reused and finally will be either improved or retired.

In this research, we have gone one step further by proposing a fully distributed architecture that is composed of many small repositories (called *sites* in this paper), distributed geographically and interconnected by the Internet. Sites can be owned and managed by organizations, divisions, departments, or even individual programmers, hence providing greater flexibility in repository configuration and finer granularity of resource sharing. A prototype system called Uranus has been developed, and techniques, such as one which used a migration history and another which is an extension of the multi-facet method [8, 9], have been developed to increase the performance of component retrieval. A user-friendly and easy-to-use interface has been designed so that a repository can be used and managed by less-experienced users.

Despite the fact that the faceted method has been criticized e.g., in [10-12], we decided to adopt this method for the following reasons. First, this method is conceptually simple, easy to implement, and operates efficiently during retrieval. Second, a browsing hierarchy can be naturally built upon a faceted classification structure. Maarek *et al.* [13] proposed an off-line method to construct a hierarchy automatically from keywords, but there seems to be no intuitive way to do so on-line. Furthermore, the browsing hierarchy built from a faceted structure is more flexible because it can be spanned in more than one way. Third, since our system is intended to be managed at the end-user level, and since tools have been developed to facilitate component indexing, the up-front cost of building repositories can be amortized and reduced. Also, note that the original method has been extended in many ways in our research.

Though conceptually simple, the design and implementation of such a distributed repository architecture is not a straightforward task. In the following sections, we begin with an overview of our system. In Section 3 our extended multi-facet classification method is presented. An important technique, migration history, is introduced in Section 4. In the next section, the internal mechanisms the query tool are described, and the implementation is described in Section 6. Finally we conclude in Section 7.

2. SYSTEM OVERVIEW

In our approach, reusable components are distributed (and replicated if necessary) to sites. These sites are interconnected with the Internet and communicate using the same protocol.

Access of read and write modes to a site can be independently secured. Each site has its own classification structure (called the *view* of that site). A user can perform a search against any site to which he/she has read access. Once the desired components are found, in addition to being reused, they can be included in the private site of the user. When adding a component to a site, the owner of the site has full freedom in deciding where in the view that component will be placed, without following the classification scheme of the source site. A component can also be put in several places in the view of a site to facilitate retrieval. In order to avoid storing multiple copies in a site, each component is tagged with a system-generated universally unique identifier (UUID), which is guaranteed to be unique in the spatial as well as the temporal sense.

The structure of the Uranus system is illustrated in Fig. 1. The *library* is a collection of components without any classification structure. UUIDs are used as the index for accessing components in the library. Classification information is stored in the view. Components in the Uranus system are classified using an extended multi-facet method, which will be explained in detail in Section 3. Both the library and the view are manipulated by the *library server*. Querying, browsing and other housekeeping tasks can be performed using several tools. When a query is performed, the *thesaurus server* and its associated *thesauri* are used to deal with synonyms. Servers and tools can be run on the same machine or on different machines. A client can communicate with several library servers or thesaurus servers at the same time, and a server can be shared by several clients.

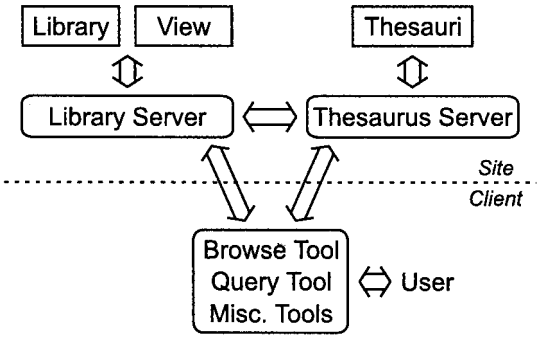

Fig. 1. System structure overview.

Clients and servers can be arranged in various configurations. For example, in the case of a personal site, the client and the two servers usually all run on the same machine. In an organization, on the other hand, a central site can be established and shared by employees. This site can be secured so that outsiders can not access it. Besides this central site, each employee can still have a private site. Finally, public sites can be

established and shared by all developers. Thesaurus servers can be configured independently.

When looking for components, a user can choose to express the requirements as a query and issue the query using the query tool to one or more sites. On the other hand, it has been pointed out [13, 14] that the existence of a browsing mechanism, which views the component collection as a hierarchy instead of a linear structure, is important for users who wish to navigate lucidly through a repository. For this reason both, a query tool and a browse tool are provided in the Uranus system.

3. CLASSIFICATION OF COMPONENTS

Our component classification method is based on the faceted method proposed in [8] and [9]. The description below, however, does not follow their terminology and notions entirely.

A *facet* is an attribute that all the involved components share. A *term* is a value assigned to a facet. A facet and its associated term together form a *factor*. A list of factors forms a *descriptor*. The property of a component can be described by one or more descriptors. These descriptors can be manually assigned to the component, or automatic techniques similar to the one used in [13] can be adopted. Components described by the same descriptor form a *class*. Note that since a component can be described by more than one descriptor, the relationship between components and classes is many-many. In a view, the set of facets is ordered according to their importance; all the components in this view share this facet set. For example, with the facet set (function, objects, medium), the descriptor (function = sum, objects = values, medium = array) can be used to describe a component that sums up values in an array. If that component can also sum up values in a linked list, it can be described by an additional descriptor (function = sum, objects = values, medium = linked list). The set of all the descriptors describing a component is called the *index* of that component. All the descriptors in a site must have the same set of facets, but different sites may have different facet sets.

A *query* has the same form of a descriptor, except that arbitrary symbols (words or phrases) can be used in place of facets or terms, and that there can be any number of factors. For example, (task = sum, objects = numbers) is a valid query that can be performed against a view with facet set (function, objects, medium), and it is likely that the descriptor (function = sum, objects = values, medium = array) will match this query.

In order to compare a query and a descriptor, thesauri are used to measure the correlation between two symbols. A *thesaurus* Θ is a function that maps a pair of symbols to the interval [0,1]. The value $\Theta(w_1, w_2)$, called the *correlation* between w_1 and w_2, indicates how closely the two symbols are related: the larger the value, the higher the correlation. For simplicity, we make a thesaurus a symmetric function, that is, $\Theta(w_1, w_2) = \Theta(w_2, w_1)$ for any w_1 and w_2. Furthermore, we define $\Theta(w, w) = 1$ for any w. A special symbol "*" is introduced as a wildcard symbol; that is, $\Theta(*, w) = \Theta(w, *) = 1$ for any w.

If a query q and a descriptor d have the same facet set, that is, $q = (f_1 = t_1, \ldots, f_n = t_n)$ and $d = (f_1 = t'_1, \ldots, f_n = t'_n)$, then the correlation between them, denoted by $\|q, d\|$, is defined as $\prod\Theta(t_i, t'_i)$. Note that because of the way Θ is defined, $\|q, d\|$ acts as a distance measure in the hyperbolic space with $f_1, \ldots, f_n$ as the axes. A missing factor in a query is

treated as if it has a wildcard term, so the query ($f_1 = t_1$) can be matched exactly to the descriptor ($f_1 = t_1$, $f_2 = t_2$). The *relevance* of a class is defined as the correlation between its associated descriptor and the query, and the relevance of a component is defined as the largest relevance of its associated classes. During retrieval, the relevance of each of the components in the view is computed, and highly relevant components are listed in order according to their relevance.

If the facet sets are different, the correlation between the query and the descriptor must be computed using a more complex algorithm, which will be introduced in Section 5.

4. MIGRATION HISTORY

One important principle behind the design of the Uranus system is that the circulation of components should be encouraged. When a useful component is found, the manager of a site is expected to download that component, or at least the component index if it is inappropriate to download the component body, from the remote site to the local site. There are several reasons for doing so. First, a new component can be indexed according to the local classification rules, so it will be easier for local users to find it in the future. Second, components can be replicated so that they will be more available. Third, the users of the local site will have a better opportunity to reach the component and distribute it to yet more sites, thus increasing its exposure to the community.

Since components can be circulated among sites, it is helpful if the user has knowledge of where a component has been. If it is revealed that a component has been residing in a site, it is likely that more components similar to it in one way or another can be found in that site. More importantly, if a large portion of the components retrieved with a query turn out to be from one site, it will be beneficial to search that site. Another purpose in tracing the past locations of a component is to find its origin. The original site may be owned by a developer which devoted to producing components of the same type as the retrieved one, so it should be searched.

As mentioned above, even in the same site, a component can reside in several classes, so we define the component migration history H_σ^c of component σ in class c as an ordered list of classes (c_1, c_2, …, c_n), where c_1 through c_n denote distinct classes. Such a migration history indicates that before arriving at class c, component σ was residing in c_n, c_{n-1}, ..., c_2 and c_1, in that order. They are called the *source classes* of σ, and c_n, in which σ was first introduced, is called the *origin class* of σ. These classes may be on different sites. In order to identify a class in a migration history, each class is tagged with a UUID, as in the case of components.

If σ is copied or moved from c to another class c', the migration history $H_\sigma^{c'}$ associated to the newly added entry in c' is defined as $c \oplus H_\sigma^c = (c, c_1, \ldots, c_{i-1}, c_{i+1}, \ldots, c_n)$ if $c = c_i$ for some i, or $(c, c_1, \ldots, c_n)$ otherwise. If σ is copied to c', H_σ^c remains the same. If σ is moved to c', H_σ^c becomes $c' \oplus H_\sigma^c$. Whether σ is moved to another class or it is deleted from c, its migration history in c is retained.

Migration history tracing can be performed manually or automatically. Automatic tracing will be explained in Section 5. Manual tracing can be done against a single component or a group of components. A set of components, which may have been

found using the query tool or the browse tool, must first be specified. Then the user can issue a command that the system list all the source classes involved, ordered according to their frequency of appearance in the migration history lists of the specified components, together with accompanying information such as the addresses of their sites and their indices. A user can then choose from among them for further exploration. For that specified set of components, the classes found by means of component tracing, no matter whether the tracing was done automatically or manually, are called *related classes*, and components in related classes are called *related components*.

The migration history mechanism can be seen as "see also" links attached to components and classes. This gives our system some of the functionality of a hypertext system.

5. THE QUERY ALGORITHM

When a query is performed against a site, if the query and the view of the site have the same facet set, the simple formula introduced in Section 3 can be used to find the relevance of a component. In other cases, things are complicated. We will first introduce the degradation function mechanism by discussing the case where the facet sets of the query and the view are identical but are ordered differently. Then, the general case will be discussed.

5.1 Degradation Functions

Since facets are specified, the order of factors in a query may be irrelevant. On the other hand, sometimes, one query factor may take precedence over another. For example, while (action = sort, object = array) and (object = array, action = sort) are equivalent most of the time, the order of factors in the query (action = sort, object = array, algorithm = quick sort) might be relevant because, usually, a user cares more about functionality than the mechanism. If no component can be found using that query, the user might be willing to accept a second best choice that matches, say, (algorithm = bubble sort). In the Prieto-Díaz system [9], a query expansion mechanism is introduced to deal with this problem. The system can, under user requests, drop the factors in the query one at a time from the tail and re-issue the query. In this case, a first-time expansion gives the query (action = sort, object = array). In our system, we introduce another mechanism called the *degradation function*. When the query ($f_1 = t_1$, $f_2 = t_2$, …, $f_n = t_n$) is matched with a descriptor ($f_1 = t'_1$, $f_2 = t'_2$, …, $f_n = t'_n$), in which the factors might have been reordered to accommodate the query, instead of the simple function defined previously, a modified function $\prod G_i(\|t_i, t'_i\|)$ is used to measure the degree of correlation. The function G_i is the ith *degradation function*, defined as

$$G_i(x) = 1 - D^{i-1}(1 - x),$$

where the constant $0 < D \leq 1$ is the *degradation coefficient*. The degradation function is a linear mapping from [0,1] to $[1 - D^{i-1}, 1]$. When $i = 1$, it is equal to the identity function. As i increases, the slope of the function decreases, thus reducing the impact due to the

fact that the corresponding query factor does not exactly match the index. This agrees with our intuition that a factor typed earlier is more important than one typed later. This method of broadening the query has three advantages. First, query expansion is done automatically; the user does not have to explicitly issue an expansion command. Second, the query system has to issue the query only once, rather than multiple times as in the Prieto-Dìaz method [9]. Third, the components retrieved after expansion can still be ordered according to their relevance. For example, since the performance characteristics of quick sort are closer to merge sort than bubble sort, the correlation between "quick sort" and "merge sort" is higher than that between "quick sort" and "bubble sort," so in the above case, the relevance of the descriptor (action = sort, object = array, algorithm = merge sort) will be higher than that of the descriptor (action = sort, object = array, algorithm = bubble sort). The query system thus can arrange the retrieved components so that the former comes higher than the latter in the list. The Prieto-Dìaz method [9] does not have this property.

5.2 The General Case

In order to deal with the general case, we must first define the correlation between two factors u = "$f = t$" and u = "$f' = t$^":

$$\|u, u'\| = \|f, f'\| \cdot \|t, t'\|.$$

Then, the correlation between the query $q = (u_1, u_2, \ldots, u_n)$ and the descriptor $d = (v_1, v_2, \ldots, v_m)$, where u_i and v_i are factors, is defined as

$$\|q, d\| = \prod_{i=1}^{n} G_i\left(\max_{1 \le j \le m} \|u_i, v_j\|\right).$$

Note that this is not a symmetric function; $\|q, d\| \neq \|d, q\|$ in general.

In the above, the correlation between two symbols cannot be found using a single thesaurus. When a query from a client to a server is performed in the Uranus system, at most three thesauri are involved. A *local thesaurus* Θ_L is used by the client to find synonyms on the client side. A *foreign thesaurus* Θ_F is used, also by the client, to translate symbols into remote vocabulary. Finally, a *remote thesaurus* Θ_R is used by the library server to deal with synonyms in the site being queried. We denote the correlation between two symbols w_1 and w_2 with respect to thesaurus Θ by $\|w_1, w_2\|_\Theta$, which is equivalent to $\Theta(w_1, w_2)$. Then, the correlation between w_1 and w_2 with respect toΘ_L, Θ_F and Θ_R is defined as

$$\|w_1, w_2\|_{\Theta_{L,F,R}} = \max_{v_1, v_2} \|w_1, v_1\|_{\Theta_L} \cdot \|v_1, v_2\|_{\Theta_F} \cdot \|v_2, w_2\|_{\Theta_R},$$

where v_1 and v_2 are arbitrary symbols.

A local thesaurus and a remote thesaurus are essential in making a query. On the other hand, a foreign thesaurus is optional. For example, a foreign thesaurus is usually

not needed when the client and the server are using the same language, say English. Only when the server is using a foreign language, or when the remote vocabulary is very different from the local vocabulary, is a foreign thesaurus needed. The user can assign a foreign thesaurus to a site. Whenever a query is made against that site, the assigned foreign thesaurus is used. If no foreign thesaurus is assigned, the correlation function becomes

$$\|w_1, w_2\|_{\Theta_{L,F,R}} = \max_v \|w_1, v\|_{\Theta_L} \cdot \|v, w_2\|_{\Theta_R}.$$

Multiple foreign thesauri is not allowed for performance considerations.

5.3 The Effect of the Migration History

At this point, we must be aware again that a descriptor is associated with a class, not with a component. Without other clue components in a class are regarded as being of equal relevance and will be (or not be) retrieved as a whole during a query session. When a class is found during retrieval, the migration history lists of its components are used to find related classes. The relevance of a related class is a function of the number of times the class occurs in the migration history lists. Suppose the history lists are $h_i = (c_1^i, c_c^i, \ldots, c_{m_i}^i),$, $1 \le i \le n$, the relevance value of a class c is

$$\alpha \cdot \sqrt{\sum_{i=1}^{n} \frac{u_i(c)}{n}},$$

where

$$u_i(c) = \begin{cases} 1 & if c \in \{c_1^i, \ldots, c_{m_i}^i\}, \\ 0 & otherwise, \end{cases}$$

and $0 < \alpha < 1$ is a constant. This relevance value is multiplied by the relevance values of the components and classes retrieved from the related class. Since a class can appear in a migration history at most once, this value is always less than one.

Using this mechanism, related classes and components can be found automatically. For example, suppose in class c, half of the components have been in another class c'; then c will be found with relevance $\alpha\sqrt{1/2} \approx 0.707\alpha$ times the relevance of c. If components in c are retrieved, it is quite likely that components in c' will also be retrieved. Consider another case where components in a class c are evenly moved to two other classes c_1 and c_2 by the site owner. This effectively removes class c, but a component's migration history is retained when it is moved out of a class, during retrieval c_1 and c_2 can still be found with relevance about 0.707α times the relevance of c. Thus, the results of a query against a site will vary only moderately even if the classification structure of that site has been altered wildly.

6. IMPLEMENTATION

A prototype of the Uranus system has been developed. Fig. 2 shows the query tool. A user can issue several queries at the same time by separating them using the "|" character. In Fig. 2, two queries, namely (function = edit, target = text) and (function = delete, target = file), are issued. The query results of those two queries will be merged together, again ordered according to their relevance, and listed in the "Results" area. Two parameters, *maximum components* (denoted by *N*) and *minimum relevance* (denoted by *L*), can be set to limit the number of retrieved components; only the first *N* components with relevance values no lower than *L* are listed. The user also has to specify the *target sites*. For simplicity, only one site can be queried at a time in the current version. During retrieval, whenever a component is found, its related classes can be checked to find more components by tracing its migration history. The user can specify a search for related classes in all other sites or some of them only. If the user chooses to trace to a number of sites only, the sites can be specified below the target site. In this example, the target site is "localhost" (the local site). and the additional sites are "gaia.ee.ntu.edu.tw" and "ibmsrv.cc.nthu.edu.tw."

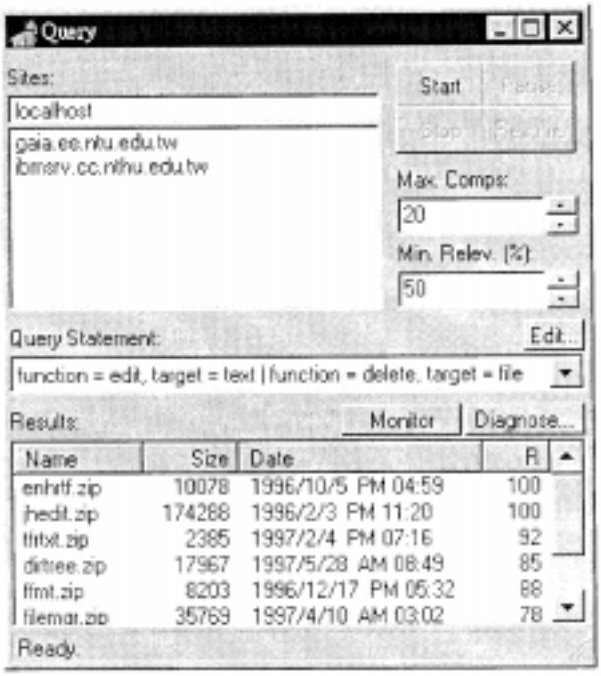

Fig. 2. The query tool.

Fig. 3 shows the appearance of the browse tool. The window is divided into three panes. To the left is the *tree pane*, which shows the classification structure as a tree. The tree structure is established by expanding the faceted classification structure along facets. This expansion is straightforward and details are omitted here. The upper-right one is the *node pane*; below it is the *component pane*. They show the sub-nodes and

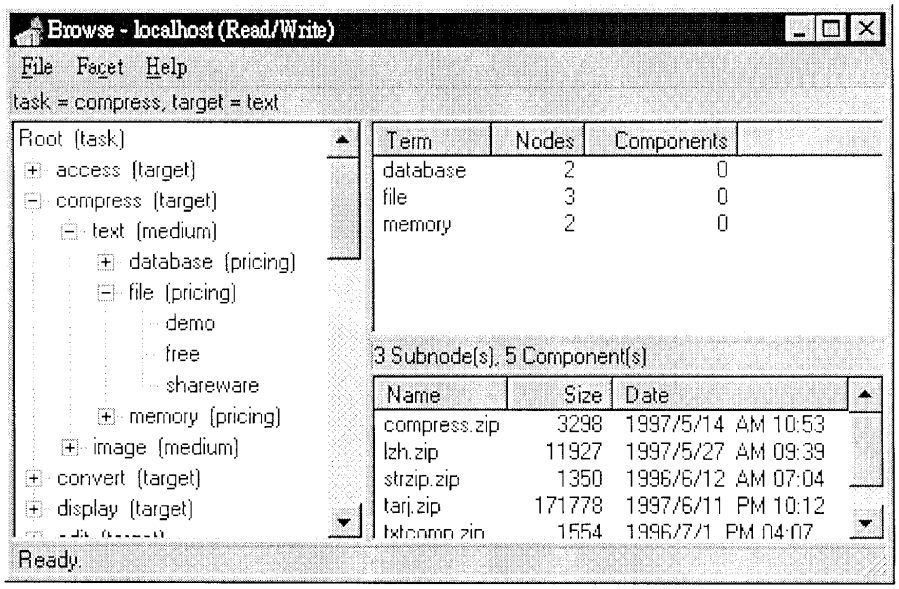

Fig. 3. The browse tool.

components, respectively, belonging to the selected node in the tree pane. The user can expand, collapse or select a tree node in the tree pane, and the contents in the other two panes will be updated accordingly.

The query and browse tools are used side-by-side to facilitate searching in component repositories. If the user has a clear understanding of the functionality of the desired component, the query tool can be used. If a component matching the requirement is found on a remote site, and if the user has the manager privilege on the local site, the found component can be added to the local site by opening the local site in read/write mode using the browse tool and then drag-and-dropping the component from the query window to a node in the browse window. The retrieved component will be associated with the descriptor of the destination node.

When the user is not familiar with the vocabulary of the accessed site, or when the requirement is not clear, the browse tool can be of greater help in searching components. In this case, the site to be searched can be opened using the browse tool in read-only mode. If the desired component is found, the local site can be opened read/write in another browse window, and the found component can then be retrieved.

In both the query and browse windows, the user can manually trace the migration history of a component to find the its related classes. With mouse clicks, the site containing a related class can be opened using the browse tool so that more components can be found. Related classes of the class corresponding to a node in the browse window can also be found.

7. CONCLUSION AND FUTURE DIRECTIONS

In this paper, we have presented our approach to developing a global development environment that enables component-oriented software reuse. Software assets can be fully distributed into small repositories that can be easily managed by end-users. The classic multi-facet classification method is adopted and enhanced with techniques, such as the use of degradation functions. A way to dynamically build and manipulate a browsing hierarchy over the faceted structure has been introduced so that repositories can be navigated and managed intuitively and lucidly. Components similar in functionality can be found efficiently using the migration history technique, even if they reside in different repositories. Finally, a set of tools with a user-friendly graphical interface has been developed to facilitate access to repositories.

In the current implementation only one view is allowed in a site. This may cause inconvenience and inefficiency if the user is willing to include a wider range of components in a site. A hybrid architecture, which allows multiple views to coexist in a site by combining enumerated and faceted approaches, has been examined and will be implemented in the future. Another possible development is the replacement of UUIDs, which are used to identify components and classes, with content-derived names (CDNs) described in [15]. Whether this is a good idea or not needs to be further investigated.

REFERENCES

1. C. W. Krueger, "Software reuse," *ACM Computing Surveys*, Vol. 24, No. 2, 1992, pp. 131-183.
2. H. Mili, F. Mili, and A. Mili, "Reusing software: issues and research directions," *IEEE Transactions on Software Engineering*, Vol. 21, No. 6, 1993, pp. 528-562.
3. W. Tracz, "Software reuse maxims," *ACM Software Engineering Notes*, Vol. 14, No. 4, 1988, pp. 28-31.
4. G. Arango, "Software reusability and the internet," *ACM Symposium on Software Reusability*, 1995, pp. 22-23.
5. S. Browne, J. Dongarra, S. Green, and K. Moore, "Location-independent naming for virtual distributed software repositories," *ACM Symposium on Software Reusability*, 1995, pp. 179-185.
6. S. V. Browne and J. W. Moore, "Reuse library interoperability and the world wide web," *ACM Symposium on Software Reusability*, 1997, pp. 182-189.
7. J. S. Poulin and K. J. Werkman, "Melding structured abstracts and the world wide web for retrieval of reusable components," *ACM Symposium on Software Reusability*, 1995, pp. 160-168.
8. R. Prieto-Díaz, "Implementing faceted classification for software reuse," *Communications of the ACM*, Vol. 4, No. 5, 1991, pp. 88-97.
9. R. Prieto-Díaz and P. Freeman, "Classifying software for reusability," *IEEE Software*, Vol. 4, No. 1, 1987, pp. 6-16.
10. M. Davis, "On practicality of domain-specific languages and analysis and multifaceted reuse libraries," *ACM Symposium on Software Reusability*, 1997, pp. 213-214.

11. S. Henninger, "Information access tools for software reuse," *Journal of Systems Software*, Vol. 30, No. 3, 1995, pp. 231-247.
12. H. Mili, E. Ah-Ki, R. Godin, and H. Mcheick, "Another nail to the coffin of faceted controlled-vocabulary component classification and retrieval," *ACM Symposium on Software Reusability*, 1997, pp. 89-98.
13. Y. S. Maarek, D. M. Berry, and G. E. Kaiser, "An information retrieval approach for automatically constructing software libraries," *IEEE Transactions on Software Engineering*, Vol. 17, No. 8, 1991, pp. 800-813.
14. H. Mili et al., "Practitioner and SoftClass: a comparative study of two software reuse research projects," *Journal of Systems Software*, Vol. 25, No. 2, 1994, pp. 147-170.
15. J. K. Hollingsworth and E. L. Miller, "Using content-derived names for configuration management," *ACM Symposium on Software Reusability*, 1997, pp. 104-109.

Yuen-Chang Sun (孫運璋) was born in Taipei, Taiwan, Republic of China, on October 5, 1969. He received the B.S. degree in Mathematics from National Taiwan University, Taipei, Taiwan, in 1992. Since 1992 he has been a graduate student in the Department of Electrical Engineering. Currently he is pursuing his Ph.D. degree. His research interests include distributed systems, programming languages, and software engineering. He is a member of the Association for Computing Machinery.

Ming-Lin Kao (高銘麟) was born in Taipei, Taiwan, Republic of China, on May 10, 1973. He received the B..S degree in Mathematics and the M.S. degree in Computer Science from National Taiwan University, Taipei, Taiwan, in 1995 and 1997, respectively. Currently he is performing military service. His research interests include programming languages, software engineering, and information security.

Chin-Laung Lei (雷欽隆) was born in Taipei, Taiwan on January 9, 1958. He received the B.S. degree in Electrical Engineering from National Taiwan University in 1980, and the Ph.D. degree in Computer Science form the University of Texas at Austin in 1986. From 1986 to 1988, he was an assistant professor of the Computer and Information Science Department at the Ohio State University, Columbus, Ohio, U.S.A. In 1988, he joined the Department of Electrical Engineering, National Taiwan University, where he is now a professor. His current research interests include software engineering, network and computer security, parallel and distributed processing, and formal semantics of concurrent programs. Dr. Lei is a member of the Institute of Electrical and Electronic Engineers and the Association for Computing Machinery.