A Multilingual Database Management System For Ideographic Languages

Chin Lu & K.H. Lee
Computer Science and Engineering Department
The Chinese University of Hong Kong, Shatin, Hong Kong

Abstract

In this project, we have built a database management system, referred to as Int_OODB, which has an internationalized user interface. Int_OODB can store text data in its original codeset. Code conversion facility is provided if the codeset used by users in his operating environment is different from the codeset in which data is originally stored. In addition, the sorting of data can be specified by a collation sequence. Our system is built in C++ with the embedded OBST object management system, a public domain software from the Stone project. We consider data entries as persistent objects which have sorting and indexing methods associated with them. The user interface is written on top of Motif(Version 1.2) under the X windows environment which supports internationalization and localization based on ISO POSIX. We localized Int_OODB for English, mainland China's simplified Chinese environment, and Taiwan's traditional Chinese environment.

1. Introduction

There is a general trend to develop internationalized software for the global market. A software needs to be ported for different regions where the language and culture conventions differs from that of the place where the software was originally developed [1,2]. Also, with the network communication available almost in every corner of the world, it is very likely to receive some multilingual documents or documents written in languages coded in non-ASCII codesets [3]. Storing data in their original codeset is very import because converted data sometimes cannot avoid loss of information [4]. Consequently, such a system must be able to access these data and display them in a codeset that the operating environment can handle and the users are familiar with.

It is desirable to store text data in a database so that the basic functions are provided by the underlying database management system (DBMS). However, most DBMSs use English-like semantics as the query language for data manipulation. Although they can be localized to users' preferred languages, text processing under database systems are still far short of support for handling ideographic characters, such as Chinese, Japanese, and Korean [2]. Usually, sorting and indexing in a database system use the internal code sequence defined in the given codeset. For alphabet-based languages, such as English and French, sorting and indexing is quite straight forward because the internal code sequence naturally reflects the alphabetic order. But, sorting and indexing ideographic characters such as Chinese based on internal code sequence does not reveal relationships among data. Therefore, browsing data indexed based on internal code sequence cannot reveal the relationships among the neighbouring data.

To handle data written in different codesets and to provide a software that can work under different languages and cultural conventions, we have built a DBMS, referred to as *Int_OODB,* which has an internationalized user interface. Int_OODB stores text data in its original codeset. Code conversion facility is provided if the operating environment codeset is different from the codeset of stored data. In addition, the sorting of data can be specified by a collation sequence other than its internal code sequence. Our system is built using C++ with the embedded OBST object management system from the Stone project. OBST is a public domain software that provides a structured and open environment for software engineering projects [5]. We consider data entries as persistent objects which have sorting and indexing methods associated with them. The user interface is written on top of

Motif [6,7] under the X windows environment which supports internationalization and localization based on the locale specifications of ISO POSIX[8]. Our focus is mainly on a DBMS capable of supporting different codesets. We have localized Int_OODB for English, mainland China's simplified Chinese environment, and Taiwan's traditional Chinese environment. Int_OODB is very useful in keeping text information in its original codeset.. It can be used in library catalogue systems, electronic library systems, and publishing and document retrieval systems.

The rest of the paper is organized as follows. **Section 2** introduces basic concepts and related work. **Section 3** presents the design principles and system architecture. **Section 4** describes design and implementation details. **Section 5** is the conclusion.

2. Background Information And Related Work

Database manipulation of Chinese text information has great demand in China, Taiwan, Hong Kong and other places where Chinese characters are used. In general, computer processing of ideographic languages like Chinese is more difficult than alphabet-based languages such as English. The complexity comes mainly from the large set of Chinese characters that a coded character set has to represent. A coded character set is often referred to as a *codeset* [9]. Chinese processing is further complicated by the fact that more than one codeset is used by Chinese in different regions. It is possible to have different encodings for the same Chinese Character.

Today, GB-2312 is the commonly used codeset in China and the CNS is the only official standard codeset used in Taiwan. The so called Big-5 codeset, which is an industry de facto standard, is also commonly used in Taiwan, Hong Kong and other places as well. In a multiple codeset environment, codeset information must be maintained if we want to keep data in its original form. We refer the machanism of maintaining codeset information as *codeset announcement*. With codeset announcement, data can be converted to others transparently without user's intervention.

2.1. Collation Sequence

For sorting and indexing of Chinese text information, four types of ordering methods, referred to as *collation sequence* [8], are most commonly used. The following shows an example of five names which yield different orders under different collation sequences:

1) By radicals first, then by strokes within radicals
   張三, 陳六, 王麻子, 李四, 趙五
2) By strokes first, then by radicals within strokes
   張三子, 六四, 麻子, 李四, 趙五
3) By pronunciation(Mainland Pin Yin)
   張三(Chen), 三子(Li), 三子陳(Wang), 三子(Zhang), 三陳(Zhao)
4) By internal code values(Big-5)
   張三子(A4FD), 三子(A7F5), 三子(B169), 三張(B3AF), 三子(BBAF)

The fourth type, which does not show any particular relationship, is the only collation sequence that most database system supports. One can argue that a particular collation sequence can be provided using secondary indexing method if a user application so requires. However, this approach is on an ad hoc basis, and it does not make the most commonly used collation sequences generally available in Chinese or other languages where sorting in internal code sequence is not desirable. A better approach is for the DBMS to provide mechanisms to install some commonly used collation sequences so that any user program can choose them at will like writing index key expressions without the need to build up secondary indexing individually.

2.2. I18N and L10N [8,9]

I18N(InternalizatioN) is the process of writing software which contains no code that is dependent on the target language, cultural conventions, and the character codeset of that language. L10N (*LocalizatioN)* refers to the process of customizing a software so that it can run under a particular language environment, referred to as the *operating environment*. To avoid changes to the source code of an internationalized software when it is being localized, the localization should be done only through changes to some tables or databases

which contain language/culture related data. For instance, a menu bar which displays the English word "Exit" should be "張三" or "張三" in Chinese. To write an internationalized software, terms like "Exit" should not be coded inside the program.

To facilitate I18N and to make L10N easy, ISO POSIX provides the specification for a set of language/culture related data, referred to as *locale*, and a set of standard functions for accessing items in a locale. Each locale defines (1) the coded character set, (2) the collation sequence if different from the internal code sequence, (3) character classifications such as *alphabet letters, upper/lower case letters*, and *control characters*, etc., (4) the set of input methods supported, and (5) cultural convention data such as date format, radix character, currency symbols, etc. POSIX also provides *message catalogues* which allow messages to be stored separately from the program code. An internationalized software will use POSIX functions to access data in a given locale which can be given at installation or run time. Whereas the localization process requires filling up the data items in that specific locale only.

2.3. OBST & X/Open

OBST is an object-oriented persistent object management system[5]. It features a hybrid data model using objects and values. It provides mechanisms to define several categories of types, the hierarchy is shown in **Figure 1.**

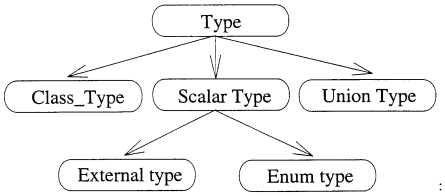

Figure 1 Hierarchy of OBST types

- *enumeration type - specify finite set of constants*
- *external type - interface OBST with the type of host language*
- *scalar type - primitive components by which classes are defined*
- *union type - for type generalization mechanism*
- *class type - user defined class type for objects*

A class type is defined by two parts: the structural properties and the behavioral properties referred to as components and methods, respectively. *Components* can be instances of a scalar type or an object. *Methods* of a class type are the entry points by which the instance of the class can be accessed. They describe the behavioral properties of the class type. There are two types of methods in OBST: definite methods and abstract methods. *Definite methods* are non-static methods that cannot be redefined in a derived class whereas the *abstract methods* must be redefined in derived classes.

OBST's data model offers *inheritance* which means that one class(*subclass*) acquires all properties of the *super class* from which it is derived. OBST also offers *multiple inheritance* in which a subclass can inherit from several super classes. Besides, OBST allows declaration and initiation of generic classes. *Generic classes* are classes which is parameterized with one or more type parameters. When instantiated, the generic class is bind to type parameters. There are some predefined generic classes of OBST which is very useful in our project (e.g. List< >, ...) and will be discussed later.

OBST provides a storage layer which uses *containers* as the mechanism for clustering and buffer management. All operations on containers are efficiently implemented by the persistent storage manager (PSM). Database objects can be stored in an individual container so that the mechanism of clustering, buffer management of each database is provided by the OBST without the need for enhancement. However, being an Object store and management system, OBST lacks support in certain common features of DMBS such as data dictionary maintenance, index building mechanisms, and database manipulation operations.

We have chosen OBST to do our pilot project because it is a free object management system running under X-window environment. Being an Object management system, OBST provides encapsulation of data. This is very important for codeset announcement because it should be encapsulated with data. When users are in the

retrieval mode to browse databases, the codeset information should be hidden from them. Also, OBST provides user redefined functions, the abstract methods, with which we can incorporate code conversion routines in the access methods which again can be provided transparently to users.

3. Design Principles and System Architecture

This project have three main objectives. Firstly, the user interface of the system must be internationalized. This ensures localization does not require any code level change. More than one operating environment should be provided to access the same database so that users can work under the environment that they are familiar with. Secondly, the system must be able to keep data coming from different sources in their original codesets so that there is no data loss as far as data store is concerned. Finally, the DBMS must have additional support for ideographic languages.

3.1 I18N User Interface

To provide an internationalized user interface, at least two issues need to be addressed. Firstly, the source code must not contain language related elements for the display of messages such as prompts, menu items, dialogue box, and on-line help, etc. These display messages must be kept in structured tables that can be accessed according to individual user's preference. Also, there are usually more than one input method associated with an ideographic language. For instance, there are around 5 to 10 commonly used input methods for Chinese input. Therefore, not only should the interface be written independent of the input methods, it should be able to handle different input methods at users requests.

The system must also allow the interface to run in an operating environment that a user is familiar with. If he is an English user, the system should display all the menus, warnings, help instructions in English. Whereas, for a Chinese user, he should have the option to choose the system that interact with him in either simplified Chinese or traditional Chinese. In fact, in the X windows environment where ISO's POSIX is supported, different users can load the same software under different locales with access to the same set of databases for different languages. In addition, for ideographic language interface initiation, the input methods associated with them must also be loaded.

3.2. Codeset Announcement and Automatic Code Conversion

To keep text data in their original codeset, the database schema definition must provide items to specify data items' codeset information, referred to as *codeset announcement* [10]. The inheritance feature in OBST can be used to provide uniform codeset announcement. Codeset announcement should be supported in two levels, one at the database level and the other at the individual record level. When announced at the database level, all text data are assumed to be written in that specific codeset. When announced at the individual record level, the announcement applies to the record only. Two levels of codeset announcement provide flexibility. For instance, if a database keeps one article in each record, a database level codeset announcement would be inappropriate if the article sources are different.. However, in a database which keeps all staff names and information of one company, it is too tedious and unnecessary to specify the codeset announcement at the record level. Codeset announcement, no matter where it is used, applies to text data only. Its specification does not affect any data types other than character string type and memo types.

Let us envision a sinario in which a user starts Int_OODB in a simplified Chinese locale, say the GB locale, and suppose that the data item he tries to retrieve and browse is coded in CNS for the tranditional Chinese. If the system does not realize that the encoding of the data item is not compatible with that of the operating environment(GB), the data item will be displayed assuming GB encoding and garbage will be displayed. With the codeset announce available, the system can recognize the difference and try to display the data in one of two different ways; One is to switch the display environment to that of the CNS, another is to convert the data to GB format before it is displayed. In this project, we choose to use the conversion approach rather than switching environment because conversion is more straight forward as long as the conversion programs (referred to as *converters* sometimes) are available. In this way, there is no

need to start another interface under a different locale.

3.3. Ideographic Language Support And Data Format

The most important issue in ideographic language support is to provide collation sequences other than internal code sequence at the system level. This is to avoid secondary indexing at the users level on an ad hoc basis for the commonly used collation sequences. Also, codesets for ideographic languages are often represented by multibyte encoding that has fixed length for each character, or by multibyte encoding that can be of variable lengths. Therefore, the source code of the DMBS not only has to be eight bits clean, it should also be able to read and write data in terms of number of characters rather than in terms of bytes.

3.4. Overall System Architecture

The Int_OODB system architecture is shown in **Figure 2**. Five major modules and components are built into Int_OODB on top of the OBST subsystem, namel, the *Internationalized Menu-Driven User Interface(I18N UI)* module, the *Ideographic Language Support & Code Conversion(ILS&CC)* module, the *B+-Tree* component, the *Data Dictionary* component, and the *Database Objects* component. The five parts are structured into three layers with the lower layers providing services to higher layers.

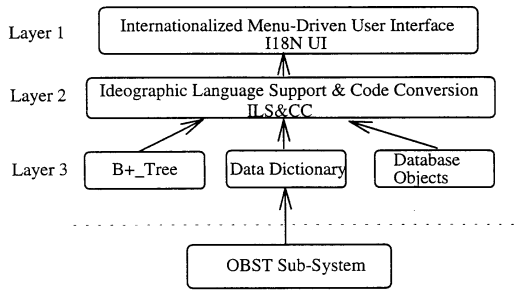

Figure 2. Overall System Design Architecture

The I18N UI module is built at Layer 3. It provides the internationalized interface for users to access the database. The 2nd layer, where the ILS&CC is defined, provides all the ideographic language support and code conversion functions including input method servers, collation sequence specifications and installation, and code conversion utilities. Layer 1 has three components: the B+-Trees for indexing, the Data Dictionary for maintaining database schema, and the Database Objects where the object-oriented databases are defined and accessed. Being an object management system, OBST lacks support in database definitions, indexing methods,

and data dictionary maintenance. Therefore, we have to build them at the lowest layer so that Int_OODB would be an actual DBMS from above Layer 1.

4. System Design And Implementation

In this section we will present the design and implementation of the five system components of Int_OODB.

4.1. Database Objects

The databases in Int_OODB system are represented by a collection of database objects. **Figure 3** shows the basic elements of a database object, defined by the class, ***DB_object***. *Root_dir* is an instance of OBST predefined persistent directory class. Through *Root_dir*, a DB_object can be referenced by its name. Whenever, a new DB_object (i.e. the database) is created, the name (of the type *sos_String*) of the object must be supplied to the Root_dir so that it can be referenced later.

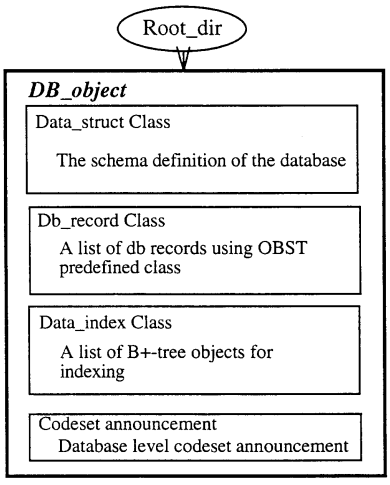

Figure 3. Components of *db_object*

A DB_object has three subclasses of objects: a data structure object of the *Data_struct* type, a list of objects of the *Data_store* type, and a list of

objects of the *Data_index* type. In addition, it has a database level codeset announcement associated with it of the *sos_String* type. the *Data_struct* object is nothing but a record to store the database schema definition Data_index class is implemented by B+_trees which will be discussed in **Section 4.2**

Each database record object of the *Db_record* type as shown in the following definition contains the actual data, the record level codeset announcement, and a **List** type provided by OBST to link to other records in the same database.

```
Db_record {
        public:
        sos_String    append_r(sos_String);
        sos_String    insert_r(Index, sos_String);
        sos_String    delete_r(Index);
        sos_String    retrieve_r(Index)
        sos_Int deleted;
        sos_String    local_codeset;
        List<sos_String>  r_list;
};
```

A record object has four methods: *append( )*, *insert( )*, *delete( )*, and *retrieve( )* for external use. Since Db_record is a subclass of the predefined *List* type, the four external methods are actually inherited rather that created anew. To minimize deletion of data at physical level, a record is considered deleted if the deleted variable is set.

4.2 B+-Trees And Data Dictionary

The *Data_index* objects of Int_OOBST are responsible for the storage of indexing structures. We used the B^+-tree structure to access data records. A B+_tree is made of nodees of four class types: *buc_ele, bucket*, *B_node*, and *B_tree*. Due to the paper length, we will not list the structures here. The relations among these classes are indicated in **Figure 4**. Basically these classes form a hierarchy in terms of building the B+_tree. The *buc_ele* class is basically equivalent to a record number. We constructed it into an object class to make uniform access to the B+_tree. The object of *bucket* holds a list of record numbers having the same index key value. The *B_node* class is the most important class among all B+_tree related classes. An instance of it is either an intermediate node or a leaf node. If it is an intermediate node, it hold keys and pointers pointing to its child nodes. If it is a leaf node, it will hold keys and the corresponding buckets.

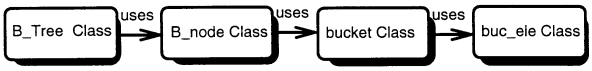

Figure 4. Relation Between Classes For B+-Tree

Figure 5 shows the internal structure of a B_node class. The *B_Tree* class is seen by application programmers for operations like insertion and search. Every instance of it contains a *B_node* which is the root node of the B_Tree. From the root node, all relevant data of the tree is accessible by the program.

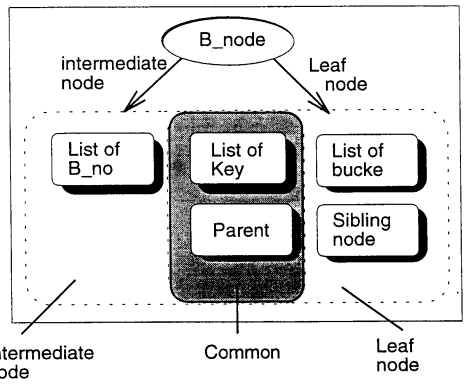

Figure 5. B_node Internal Structure

The data dictionary component is an addition to the OBST system to provide directory services for users to query about databases that exist in the system. Our data dictionary component offers three services. Firstly, it maintains a name list of database created. Secondly, it maintains index object(s) created for each database. Thirdly, it provides directory service which allows databases to be organized in a hierarchical way.

Figure 6 shows the structure of the data dictionary. A *Data Dictionary* object is composed of Dictionary Entries. Each entry contains dictionary information of one database object including its name and the indices defined under it. The index object not only points to the B+_tree of this index, it also keeps the index expression that created this index so that such information can be obtained later.

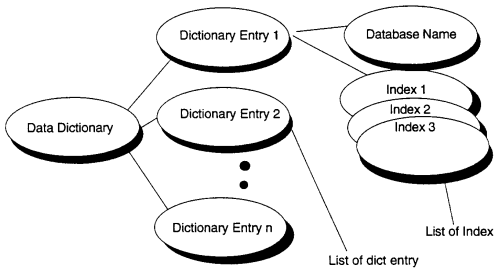

Figure 6 Data Dictionary Structures and Class Relationships

4.3. Ideographic Support

The mechanism for defining collation sequences is provided by POSIX under the definition of locale [8]. In order to use a collation sequence other than the internal sequence, the environment variable *LC_COLLATE* must be set. POSIX specification allows only one collation sequence to be defined under each locale directory. To support different collation sequences, we must create different locale directories with the same set of files except the collation sequence file. An index created to use the collation sequence must use POSIX function to obtain the value for each key before the key can be inserted into the index tree. When sorting data based on a specific collation sequence, the order is set according to the POSIX conforming function *strcoll( )* rather than *strcmp( )* in ANSI C. Int_OODB aims at providing a facility so that the commonly defined collation sequences can be defined. When a collation sequence is provided in the system, we can easily incorporate it into the Int_OODB. The provision of collation sequence is a part of the localization process. In this project, we have built the Chinese collation sequences based on radical, spelling and number of strokes.

In setting the proper operating environment for ideograph language users, we have also built our input method handling mechanism based on locale [11]. That is, when you start the system by specifying a particular locale, the input methods associated with that locale will be loaded automatically. The input methods are handled by an input method server that can serve applications in different locales at the same time. When you start Int_OODB more than once, possibly in different operating environments, the input methods will be handled by one server similar to X window servers. Due to the scope of this paper, details of the input method server will be described in a separate paper.

We have used a code conversion interface developed by the Hanzix group to provide automatic code conversion. mechanism. Code conversion is needed in the following three situations when

- Retrieving data from a database whose codeset is different from the operating environment's codeset
- Storing new/updated database items back to the database whose database /record level codeset is different from that of the operating environment
- Retrieving directory/database names from the data dictionary

Three functions *hz_iconv_open( )*, *hz_iconv( )*, and *hz_iconv_close( )*. are the core provided in the code conversion interface. Basically, a code converter is a program that takes a string as input and returns a converted string of the specified codeset. Before any conversion can be done, the converter has to be loaded first by using *hz_iconv_open( )*. The actual conversion is done by *hz_iconv( )*. When conversion is no longer needed, the converter can be closed by *hz_iconv_close( )*. The conversion functions are built into the database abstract methods. Therefore, its implementation is completely transparent to users. Automatic code conversion is provided according to the locale of the user's operating environment and the code announcement mechanism provided in Int_OODB. One thing that needs to be pointed out is that character conversions sometimes may have a one-to-many mapping. This can sometimes cause problem especially if the conversion is done for storing data items back to the database because in this case the so called original data format is not original anymore. Dealing with one-to-many mapping is out of the scope of this project. However, the Hanzix code conversion interface does provide mechanism to load different converters, some of which can be intelligent, thus avoid the one-to-many mapping problem. Int_OODB can use this mechanism to load a converter of user's choice by specifying it in a resource file used when starting up the user

interface. We do encourage people to update a database in an environment that is not consistent with its codeset announcement. But the system does not enforce it.

The *date* type which is supported as the basic data type in most DBMS must take into account of different cultural conventions. Different countries and regions use different formats. For instance, U.S. uses mm/dd/yy format whereas Hong Kong uses dd/mm/yy format. To make sure that data can be stored and interpreted correctly, we have fixed all *date* type data to be stored in the YYMMDD format. The conversion to its right format for output is done through the locale specification by obtaining the LC_DATE format.

4.4 Internationalized User Interface

The user interface is implemented using Motif under the X Windows environment. The I18N of the user interface is accomplished by completing three parts. Firstly, we have separated the program logic from program messages by using calls to *catopen()*, *catgets()* and *catclose()* to retrieve message text at run-time rather than hard-code messages into the source program. Secondly, we have created a set of message source files for localization. For Chinese localization. We wrote only one set of message source files for one locale and use the Hanzix code conversion routines to port it to a different Chinese locale. Thirdly, we have generated a message catalogue from the message source file using *gencat,* a program that comes with the NLS(National Language Support) package of X window.
The message files are written in a very similar way as *abc.msf* with *cxterm,* a Chinese terminal emulator for X window system [12]. The text source file has a suffix *.msf*. Each message is associated with an integer as an index and each group of Chinese messages can be grouped together with a set number. For example,

$set 1
1 張三
2. 張三
$ set 2
1. 張 子
2. 張 子

After the message catalogues are generated using the *gencat* program, they should be located at the respective locale directories. For instance, the GB message files should be put in directory *~/nlsmsg/zh* whereas the CNS message files should be put in *~/nlsmsg/zh_TW*. The name of the language(codeset) and the path for the message files should be set in the environment variables *LANG* and *NLSPATH*, respectively. The following is an example of how to set it for the GB locale:

setenv LANG zh
setenv NLSPATH ~/nlsmsg/%L/%N

where *%L* is a variable to represent the environment variable *LANG* and *%N* is a variable to represent the catalogue names specified in *catopen( )*. A catalogue is opened by the function *catopen( )* with the following syntax:

nl_catd catd catopen ("abc.cat", 0)

where *catopen( )* returns a nl_catd as a file handler for future use. The second argument of *catopen( )* is reserved for future use and its default value is 0 up to now. The messages are retrieved by the function *catgets( )* with the following syntax:

*char * catgets (catd, 1, 1, "Hi")*

where the first argument is a nl_catd, the second argument is a set number and the third argument is the index of the Chinese message. The last argument is used as the default return value in case the specified message does not exist in the catalogue.

Figure 7a and **Figure 7b** show two instantiations of the Int_OODB with different LANG settings. In **Figure 7a**, **t**he value of LANG was set to zh, the name of mainland China's GB locale. In **Figure 7b** the value of LANG was set to zh_TW Taiwan's CNS locale. Notice, that the difference between the two is in the way that Chinese is displayed. The database opened on the screen is in fact the same. Conversion of data is done automatically.

Figure 7a Interface for *zh*

Figure 7b Interface for *zh_TW*

5. Conclusion

In this project, we have implemented an internationalized database system with additional support for ideographic languages. Our focuses is on supporting different Chinese environments, but the system can also be extended to support other ideograph languages.

Through the development process, we have realized that internationalization can improve productivity and save time for localizations. However, under the POSIX specification, multilingual support is limited to two languages in one application, namely, English(or any other alphabet based language) plus another language, such as, Chinese. It does not provide a general multilingual environment in the sense that an arbitrary number of languages can be supported in one application. Of course, we can always generate some encoding method in a particular application that can switch among different languages, but it does not provide a general solution and it may not be compatible with the underlying system support. We are currently working on research that can truly support multilingual applications. One of the directions we have looked at is the used of 10646 codeset.

References

[1] Buhle E.L. Jr., "Writing International Code", *Digital Systems Journal*, Vol 16, Iss 3, May, 1994

[2] H. Yoshioka and J. Melton, "Character Internationalization In Databases: A Case Study", *Digital Systems Journal*, Vol. 5, Iss 3, May, 1994

[3] R. Stokes, "New Information Technology: Acquiring, Processing, And Accessing Resources in Asian Languages", *Australian Library Review*, Vol. 11, Iss. 3, Aug, 1994

[4] Hanzix Work Group, "The Hanzix Open System", *Proceedings of International Conference on Chinese Computing '94,* 1-4 June 1994, Singapore

[5] Michael Ranft, Walter Zimmer and Jochen Alt, ″OBST Release 3.3 User Guide and Manual″ , March 1993

[6] Adrian Nye and Tim O' Reilley, "The Definitive Guides to the X window System Volume Four : X Toolkit Intrinsics Programming Manual OSF/Motif 1.2″, Motif Edition, *O' Reilly & Associates, Inc.*, 1994

[7] Dan Heller & Paula M.Ferguson, ″The Definitive Guides to the X Window System Volume Six A Motif Programming Manual for OSF/Motif Release 1.2″ Motif Edition, *O' Reilly & Associates, Inc.*, 1994

[8] ″Introduction to XSI Internationalization″, *X/Open Portability Guide,* Volume 2., chapter 2-8.

[9] Sandra Martin O'Donnell, "Programming for the World: A Guide To Internationalization", *Printice Hall*, 1994

[10] Hanzix Work Group, "Hanzi Codeset Conversion and Announcement", *Proceedings of the 1992 International Conference on Chinese Information Processing,* Beijing, Oct., 1992

[11] Hanzix Work Group, "Open Systems Environment For Hanzi Input Methods", *Computer Processing of Asian Languages '92*, Kanpur, India, March 12-16, 1992

[12] Man Chi-pong and Yongguang Zhang, "Cxterm: A Chinese Terminal Emulator for the X window System", *Software practices and experiments*, Vol. 22, Iss. 10, October 1994