1998 International Computer Symposium
Workshop on Software Engineering and Database Systems
December 17-19, 1998, N.C.K.U., Tainan, Taiwan, R.O.C.

DESIGN AND IMPLEMENTATION OF VERSION CONTROL FOR THE QBOE MULTIMEDIA DATABASE SYSTEM

Ye-In Chang and Hong-Nian Chen

Dept. of Applied Mathematics
National Sun Yat-Sen University, Kaohsiung, Taiwan, R.O.C.
E-mail: changyi@math.nsysu.edu.tw

ABSTRACT

In advanced applications, especially those in which a database has to support the design of a manufactured product, *versions* of objects have to be managed; that is, to keep different status of the same data, which is called *historical versions*. This requirement is inherent in applications that are exploratory and evolutionary. Version control in the multimedia document creation environment is similar to that in any design environment. In this paper, based on the object-oriented data model, we design and implement version control for a Query-By-Object-Example multimedia database system, in which we consider data of types text, drawings, images, audio, animation and videos. The system and the query language are both called Query-By-Object-Example, since the user-interface is through an object example.

1. INTRACTION

In traditional DBMS, once transaction updates have been committed and permanently installed, the previous values of data usually are discarded. However, advanced multimedia applications, especially design applications, require facilities to maintain data versions; that is, to keep different status of the same data, which is called *historical versions*. This requirement is inherent in applications that are exploratory and evolutionary. Version control in the multimedia document creation environment is similar to that in any design environment. Moreover, versions can be used for different purposes, including concurrency control, recovery, enhancing performance, and implementing "update-free" [5].

There are three implementation issues in version control: when to create a new version, how to represent the version, and which version in a database represents a consistent configuration of objects [6]. (When the concepts of a version history and component hierarchies are combined, the result is a *configuration*.) The third issue contains how to connect all versions of the same data and how a version reference is executed. A *static* version reference strategy is much simpler than a *dynamic* version reference strategy; however, a *static* reference strategy may refer an old version of data, while a *dynamic* reference strategy always refers the newest version of data. How to provide mechanisms to manage historical versions is an important research topic.

In this paper, based on the object-oriented data model, we design and implement version control for a Query-By-Object-Example multimedia database system, in which we consider data of types text, drawings, images, audio, animation and videos. The system and the query language are both called Query-By-Object-Example, which is denoted as QBOE, since the user-interface is through an object example. Basically, in order to maintain a consistent configuration of objects, we have to consider the effect of different operations to the versions, including deleting or updating a sharing object, change notification, change propagation and inheritance. Figure 1 shows an example of a multimedia resume, which is used throughout the paper.

2. DDL and DML

In this Section, we present our QBOE multimedia database system. We first describe our design of DDL. Next, DML is introduced. For convenience, we use the example shown in Figure 1 to illustrate the processes of DDL and DML.

2.1 DDL

When a user enters the system, there are two options: *query* and *create*. To define a Resume class, a user first chooses the *create* option. Next, the system asks the user to enter an object name. In this example, it is *Resume*. For each object, the system asks for more information: *Class Type*, *Part Number*, *Data*, *Attribute*, *Method* and *Versionable* as shown in Figure 2.

Those fields must be filled with an integer, which

[1] This research was supported by National Science Council of the Republic of China, NSC-86-2213-E-110-004

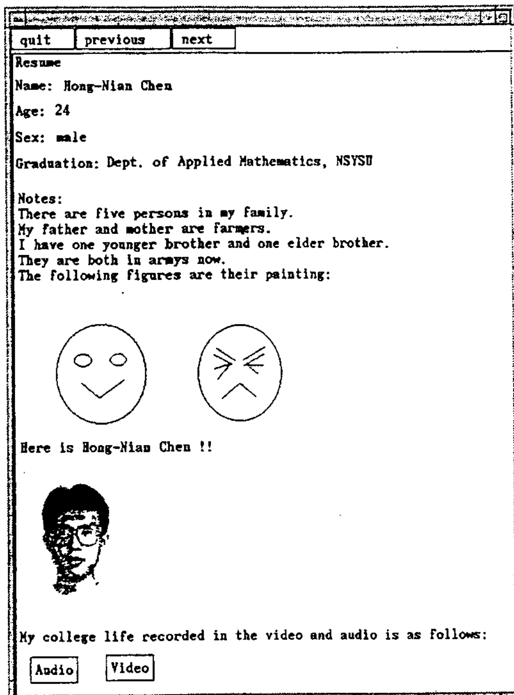

Figure 1: An example of a multimedia Resume

means the number of parts of related information or with a character "Y" or "N" which means "Yes" or "No". For example, in Figure 2, the user enters 2 in the *Part Number* field, which means that the Resume object contains two parts. The user also specifies that the Resume object is versionable at the the bottom field by entering a character "Y". The system then asks the user to input two names of children one at a time. For the Resume object, the user can also specify some attributes. For example, the Resume object has a Font-Size = 12 attribute.

Since in a standard Resume object, there are always some keywords like *Name*, *Age*, *Sex*, and *School*, we also enable users to input these keywords as built-in data for each Resume object. First, the user specifies the number of built-in data in the Data field. After that, the user can input a file name as the input data. If the file exists, the system loads the data stored in the file and shows it in the screen. If the file does not exist, the system asks the user to choose the type of data, Text, Drawing, Image, Audio, Video, Animation, and it invokes a related editor.

For some object, there may be some related methods associated with it. For example, there is a Draw method for the Drawing object. Therefore, after specifying the number of methods in the Method field, in a similar way, the user can input a method name for

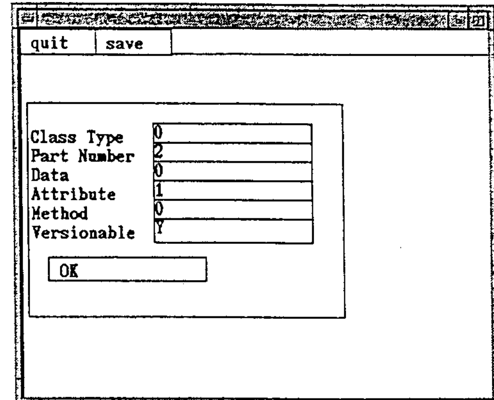

Figure 2: Information about an object Resume

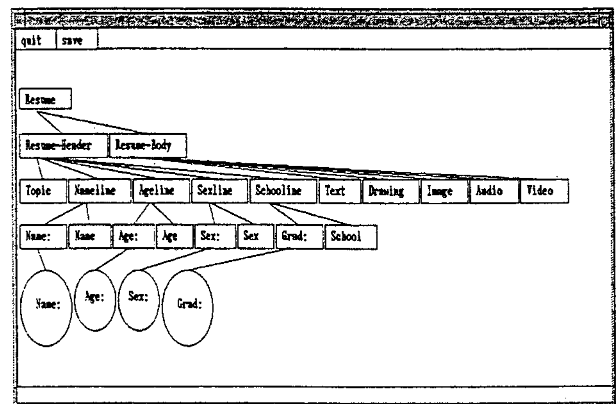

Figure 3: Final structure of object Resume

the Drawing object. Moreover, an object can be versionable or nonversionable, which can be specified in the Versionable field. If an object is nonversionable, its children are also nonversionable. Note that an update to a versionable object creates a new version of this object, while an update to a nonversionable object overwrites the old data. Finally, Figure 3 shows the final structure of object Resume, where the circle nodes mean the data that has been created by DDL. Every instance of the Resume class shares these data objects.

2.2 DML

We have described the *Create* function in the previous section and have created a new Resume schema. Next, we show how the other function – *Query* – works. After we choose the Query function, the system pops up a window which shows all of the functions that the system provides: *Insert*, *Delete*, *Select* and *Modify*.

After we choose the *Insert* function, a pop-up window appears, which asks the user to enter an ob-

1998 International Computer Symposium
Workshop on Software Engineering and Database Systems
December 17-19, 1998, N.C.K.U., Tainan, Taiwan, R.O.C.

ject class name. After Resume is entered, the system returns the Resume class hierarchy. After the *addNewOne* function is chosen, two windows pop up at the same time. The upper one is called the *control table*, and the lower one is called the *output window*. The *control table* shows the objects that can be edited in the output window at this time.

After we choose the *Resume-Header*, the objects shown in the *control table* is replaced by the children of the Resume-Header. At this time, a new rectangular box shows up in the output window, which represents the positions in which any of these five objects: Topic, Nameline, Ageline, Sexline and Schooline, can be placed. The rectangle can be moved to any place in the output window.

After we move the rectangle to a certain position, we can start to enter data. For example, after we choose the Topic object, a pop-up window that inquires the user to input data appears again. Then, another window appears that asks the user to input the file name. After *Topic* is entered, the system asks for the data type of the specified file name. In this case, Text is chosen. Assume that the *Topic* file dose not exist; then, a text editor, "vi," is invoked. After we edit the file and exit "vi," the file is ready and is copied to the output window. Next, the system asks the user to input a keyword. (Note that if the *Topic* file already exists, the contents of the file shows up in the output window without invoking the text editor "vi.") Finally, Figure 4 shows the output window after the insertion is finished. This is the example that we have used throughout the paper.

For the Modify function, there are four options: *modify, move, insert* and *delete.* Those four options operate on a single object at a time. When a user modifies a versionable object, the system will create a new version. Before data modification, the user must first find out what the user wants to update.

Let us consider the case of data selection. First, the system asks for an object name. Then, the object hierarchy appears, and two selection options, *contain-based* and *keyword-based*, are shown in the upper left corner. After the user chooses the Name object, the word Name appears on the bottom line of the screen. In this example, the user types "Chen" in the box of the upper left corner. After that, a sentence (Name CONTAINS "Chen") appears on the bottom line of the screen. Now, the user has entered the condition that is equivalent to an SQL statement (select * from Resume where Name CONTAINS "Chen"). Since at this point, such an object has only one version, the only one version is shown up.

Next, let us see an example of a keyword-based

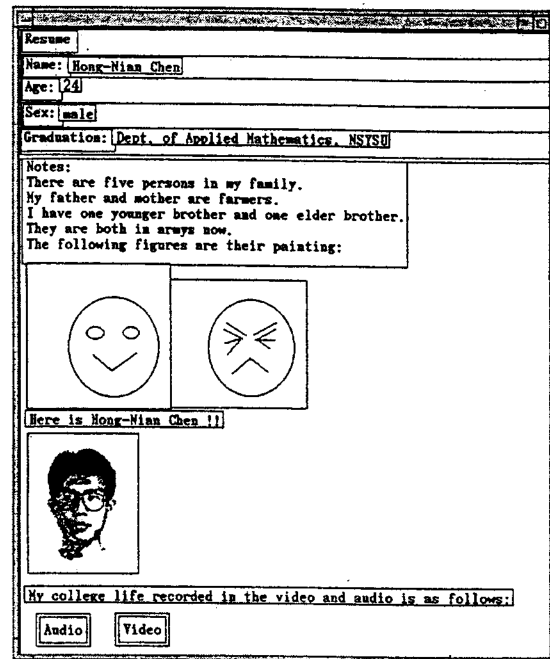

Figure 4: The output window after the insertion is finished

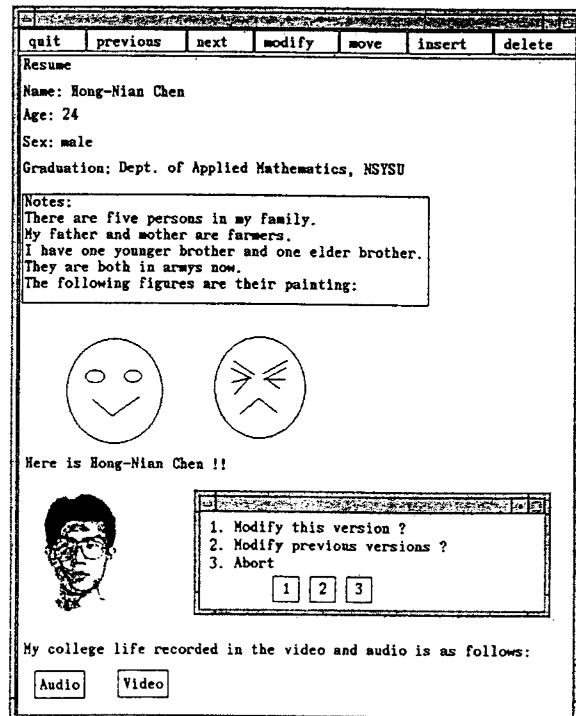

Figure 5: An example of modifying a versionable text object (Version 1)

query. After the user chooses the Drawing object, the word Drawing appears on the bottom line of the screen. The user then chooses the *keyword-based* function. At that time, an empty box appears on the upper left corner. In this example, the user types "pain" in the box. After that, a sentence (Drawing: KEYWORD = "pain") appears on the bottom line of the screen. Now, the user has entered the condition that is equivalent to an SQL statement (select * from Resume where Drawing: KEYWORD = "pain"). The same result is obtained.

After the data shows up, the user can move the cursor to the part which he (she) wants to modify by choosing the *modify* option. After the user clicks on that part, the related window appears. Take the Text window as an example. Since the text object is versionable that have been defined in class Resume, the system pops up a window with three options to choose the suitable version (either current one or previous one if it is available) or abort it. In this example, since there is no previous version, the current version is chosen, as shown in Figure 5. Then the user can enter the text editor "vi" environment to modify the text part.

After the user exits the text editor, the system asks the user to confirm this change. If the answer is "Change", then the updated file is copied to the output window. At this time, a new version history for this text object has been created. Similar to the update to the text object, the update to a drawing object is shown in Figure 6. Again, the system creates a new version hierarchy. When the user confirms such a modification, a new configuration for object Resume is created.

Next, we enter the Modify option again and select the second version. This time, we change the text part again, and a new version history for this text object has been created. Then, we change the text file one more time, and a new version history for this text object has been created again. Similarly, an update to the drawing object occurs again. In the case, the user uses *version 1* of the drawing object to replace the original version as shown in Figure 7. As described before, if the user confirms such a modification, a new configuration for object Resume is created again. Figure 8 shows the relationship of the configuration and the version history hierarchy after the above modifications. Note that, in Figure 8, the system creates a new version 1831.3 of object Resume and a new version 1846.3 of object Resume-Body after the user exits the Modify option.

After that, we select the second version of object Resume and modify it again. When the *delete* option is chosen, the selected object is deleted one at a time. When the *move* option is chosen, the selected object can be moved to any place in the output window. When the *insert* option is chosen, a new object can be inserted. Then, the system asks the user to choose the type of the data as following the steps of the Insert function described before. The related editor is invoked if it is necessary. Figure 9 shows the result of this update. When the user confirms such a modification, a new configuration for object Resume is created.

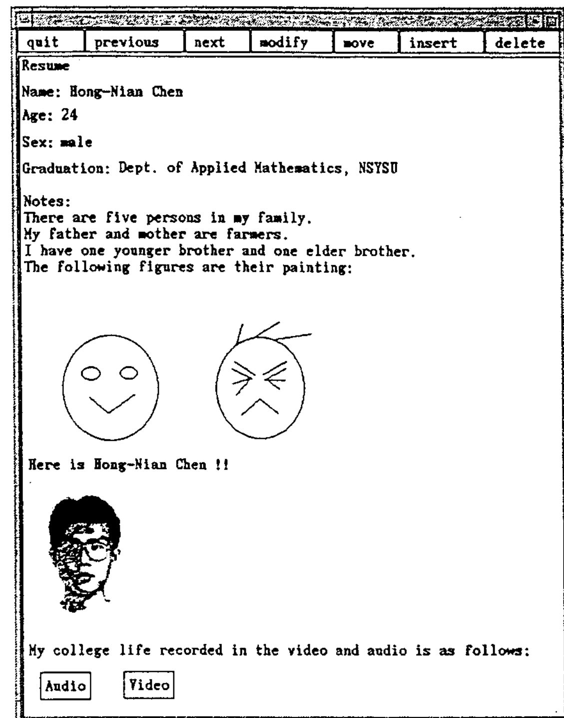

Figure 6: An example of modifying a versionable drawing object (Version 2)

In the Select option as described before, an object can be chosen by either *contain-based* or *keyword-based*. After the user chooses the object Name by following the *contain-based* Selection, the word Name appears on the bottom line of the screen. In this example, the user types "Chen" in the box. After that, a sentence (Name CONTAINS "Chen") appears on the bottom line of the screen. Now, the user selects the *run* option. Then, the system pops up a window with five options as shown in Figure 10. One option is *Default* and the others are *Version 1*, *Version 2*, *Version 3*, and *Version 4* since we have created 4 versions for object Resume, where Default means that the object is accessed by dynamic reference, while the other four options mean that the object is accessed by static reference.

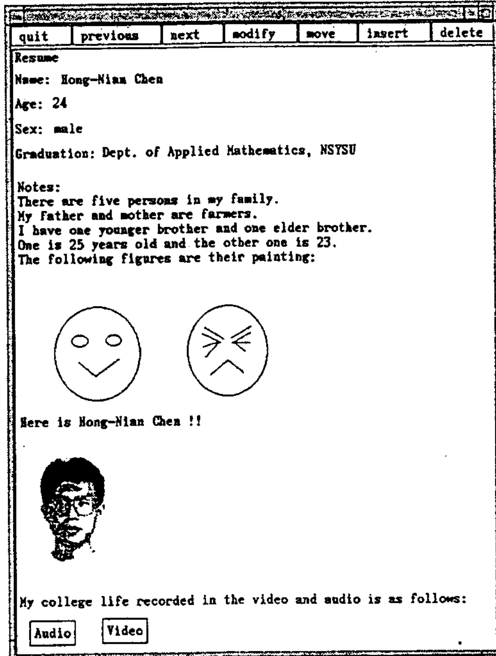

Figure 7: An example of modifying a versionable drawing object (Version 3)

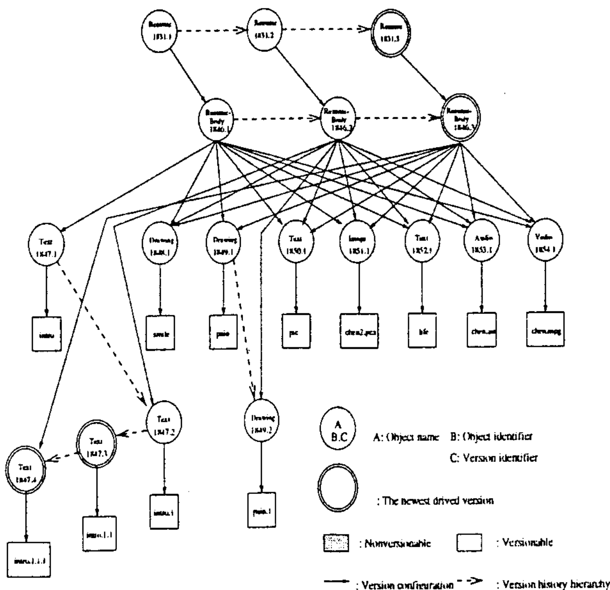

Figure 8: The details of the modification part (Version 3)

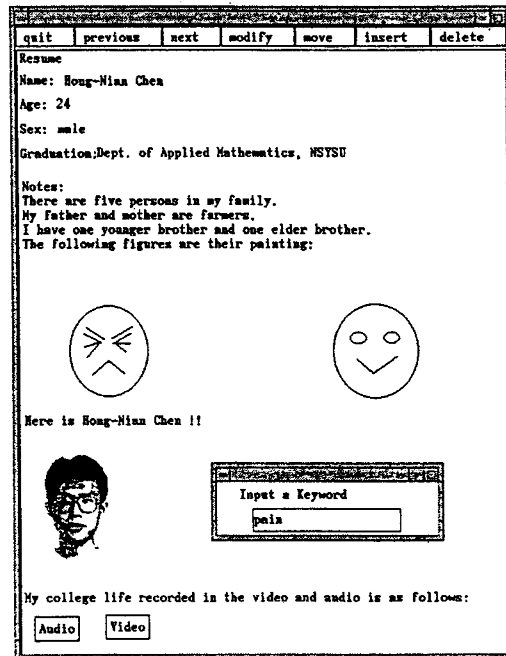

Figure 9: An example of inserting a new object (Version 4)

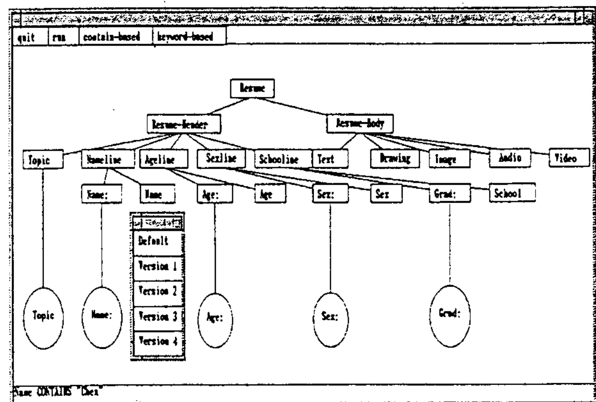

Figure 10: A version selection menu

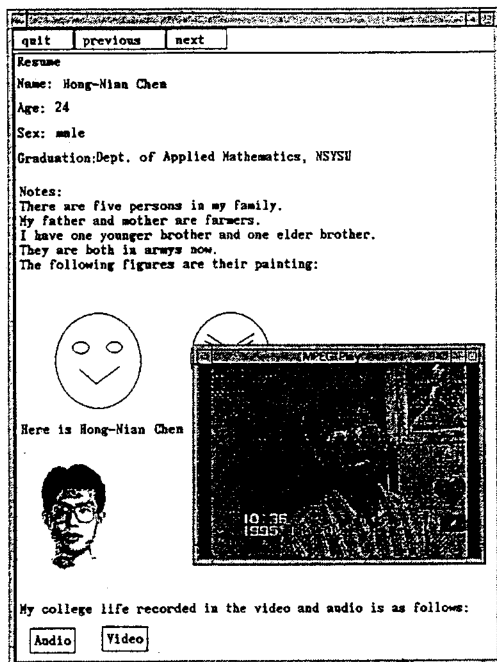

Figure 11: An example of selecting a video object

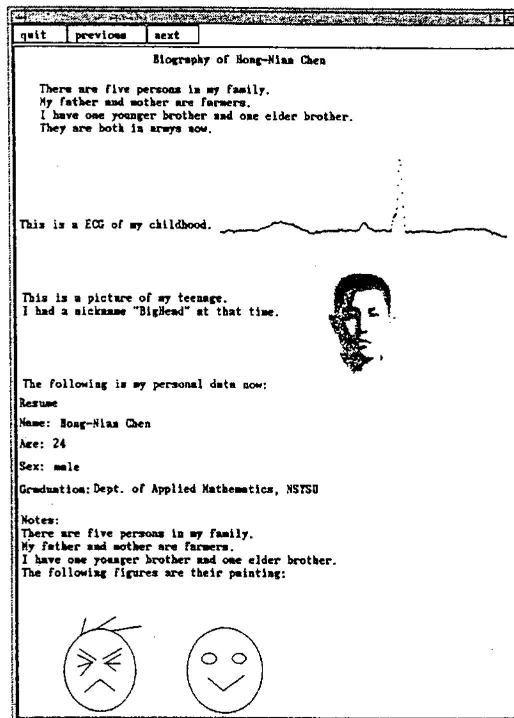

Figure 12: An example of the Biography object which contains version 2 of the short Resume object

To implement the SQL statement (select Text from Resume where Name CONTAINS "Chen"), the user can move the cursor to the Text part which the user is interested in and click on the button one time, which results in a new Text window. In the same way, the user can move the cursor to the other part (Drawing, Image, Video, Audio), and click on the button one time to open a new window. Figure 11 shows the result of the SQL statement (select Video from Resume where Name CONTAINS "Chen").

For the Delete function, the user follows the method for using the Select Function to find out what the user wants to delete. After the data shows up, the system asks the user to confirm the user's choice.

Let us see one more example, a Biography object, which is a composite object including object Resume. To simplify our following presentation for the operations to the composite object, Biography, we make two short versions of Resume object. When we first insert data for object Biography, we choose *Version 2* as its member. The resulting Biography object is shown in Figure 12. We treat object Resume like a normal object. When *Version 2* of the short Resume object is changed to *Version 3*, we have a second version of object Biography automatically.

Note that in this case, when a deletion operation happens on any one of the versions of the short Resume object that is referred in the Biography object, the system will reject such a deletion operation. While a deletion operation on a version of the short Resume object through the Biography object is allowed. Moreover, when *Version 1* of the short Resume object is changed to *Version 4*, the system does not create a new version for the Biography object. That is, only updates on *Version 2* of the Resume object, or on the children of *Version 2* in the version history hierarchy will result in a new version for the Biography object automatically. Furthermore, an update to *Version 3* of the short Resume object through the Biography object also creates a *Version 4* of the short Resume object automatically.

3. IMPLEMENTATION

Our system is implemented using C^{++} and X-lib [7]. Figure 13 shows the logical implementation of an object-oriented data model described in Section 2. This structure defined in C^{++} is shown in Figure 14. Note that there are six additional elements, *isTypeOfCount*, *partCount*, *dataObjCount*, *attrCount*, *methodCount*, and *versionable*. We use these first five elements of the *CLASS* structure to record the variable parts of the CLASS structure – *isTypeList*, *partList*, *dataObjList*, *attrList*, and *methodList*, respectively. The last element of the *CLASS* structure is to record whether

obj-id	class-name	is-type-of	can-have-parts			has-data	has-attributes	has-methods	versionable
		obj-id	min	max	obj-id	obj-id			

Figure 13: Logical data structure of a class object

```
struct CLASS{
int objectID, isTypeOfCount, partCount, dataObjCount, attr-
Count, methodCount, versionable;
char *className;
int *isTypeList;
PART *partList;
char *dataObjList, **attrList;
int *methodList;
CLASS *next;};
struct PART{ int min, max;
CLASS *object;
char *objName;
int objID;};
```

Figure 14: The C^{++} implementation of a logical class object

this class object is versionable. All the class objects that have been created are saved in the *classFile*. For example, in one of class objects recorded as

7612 0 2 0 0 0 1 Nameline 1 1 3005 1 1 93

the 3rd column has a value of 2. This indicates that the Nameline object has two children. Next, the 7th column has a value of 1. It indicates that the Nameline object is versionable. The values following Nameline describe the children's characteristics. The first number and the second number indicate the minimum number and the maximum number of the occurrence of the first child of this class, respectively. The third number is the ID of the first child.

Figure 15 shows the data structure of an instance of a class object. All the instances of Resume class are stored in the instFile file. For example, in one of instances recorded as

1836 0 1 5 0 1 1 93 Name Chen type=1 wx1=57 wy1=2 wx2=189 wy2=20

the third column has a value of 1, which indicates that the object has a data file whose name follows the class identifier 93, i.e., *Name*. The 6th column has a value of 1, which indicates that the object has a keyword whose content follows the data file name, i.e., *Chen*. The 4th column row has a value of 5, which indicates that the object has 5 more attributes to be recorded, i.e., *type*, *upperleftX*, *upperleftY*, *lowerrightX*, *lowerrightY*. Following the keyword content, there is a string *type* $= X$, which indicates that the data is of the type text, drawing, image, audio, video, animation and none of the above for $X = 1, 2, 3, 4, 5, 6, 0$, respectively. The next four strings record the coordinates of the upper left corner and the bottom right corner of the data shown in the output window.

Next, Figure 16 shows the data structure of a version history hierarchy of an object. The first two elements, *objectID* and *versionID*, are used to record object identifier and version identifier, respectively. The element, *dataCount*, is used to record whether this version object contains a real data file. If *dataCount* equals to 1, *versionName* contains a version file name. That is, this object is a leaf object in the class hierarchy. If *dataCount* equals to 0, it means that this object has parts. The rest of the following elements are used to record the version history, configuration hierarchy and location. A version history is recorded by *versionCount* and *versionList*. A configuration hierarchy is recorded by *partCount* and *partList*. Elements *upperleftX*, *upperleftY*, *lowerrightX* and *lowerrightY* record the location of this version object on the document.

For example, in one of the version objects recorded as

1847 2 1 1 0 intro.1 3 3 1 450 122

the first two columns has values of 1847 and 2, which denote the second version of object 1847. The 3th column has a value of 1, which indicates that this version object is a real data file. The 6th column is the corresponding data file name, *intro.1*. The 4th column has a value of 1, which indicates that this version object has a child version, 3, which is recorded in the 7th column in the version history. The rest of elements are the location of this version object. Take another example, in one of the version objects recorded as

1831 2 0 2 2 3 4 1832 1 1846 2 1 1 651 836

it records the version information about the second version of object 1831. The 3th column has a value of 0, which indicates that this version object has an aggregation relationship, i.e., it has parts. The 4th column has a value of 2, which indicates that this version object has two child versions. The 6th and 7th columns record the child versions. The 5th column records the number of parts and the related object identifiers and version identifiers for those parts are recorded in those columns following the 7th column, which are the first version of object 1832 and the second version of object 1846.

Figure 17 shows the data structure of *RefCount*. That data structure helps us to record the relationship between a referencing class and a referenced class, and make us to handle the deletion operation easily. The first two elements are used to record root object identifier and version identifier. The third element, *Count*, is used to record that how many times this root version object is referenced. Figure 18 shows the relationship

```
struct ObjInst{
int objectID, partCount, dataCount, attrCount, methodCount,
keywordCount, versionCount, classID, *partList, *firstPartEn-
try;
ObjInst **partEntry;
char **dataList, **attrList;
int *methodList;
char **keywordList, *type, *upperleftX, *upperleftY, *lower-
rightX, *lowerrightY;
ObjInst *next;};
```

Figure 15: The data structure of an object instance

```
struct ObjVersion{
int objectID, versionID, dataCount, versionCount, partCount;
char *versionName;
int *versionList;
childVsn *partList;
int upperleftX, upperleftY, lowerrightX, lowerrightY;
ObjVersion *next;};
struct childVsn{ int childID, versionID;};
```

Figure 16: The data structure of a version history hierarchy of an object

between the class Resume, denoted as A, and the class Biography, denoted as B. Whenever a class A is included as a data part of a class B, the relationship between A and B is established in this structure and *Count* is increased by one. When the version or any child version of class A that is referenced in class B is updated to a new version, a new version for class B will be created automatically by inserting a new entry into the *versionFile* file and *A.3->Count* is increased by one. When class B is updated to a new version in which class A is updated, a new entry for classes A and B will be inserted into the *versionFile* file, and *A.4->Count* is increased by one. As described before, when a deletion operation happens on any one of the versions of object A that is referenced in object B, the system will reject such a deletion operation. While a deletion operation on a version of object A through object B is allowed. When a deletion operation on a version of object A through object B occurs, *A->Count* is decreased by one. Only when $A.X.Count = 0$, version X of object A is deletable.

```
struct RefCount{
int objectID, versionID, Count;
VersionObj *refObj;
RefCount *next;};
struct VersionObj{ int objectID, versionID;};
```

Figure 17: The implementation of a class referenced count

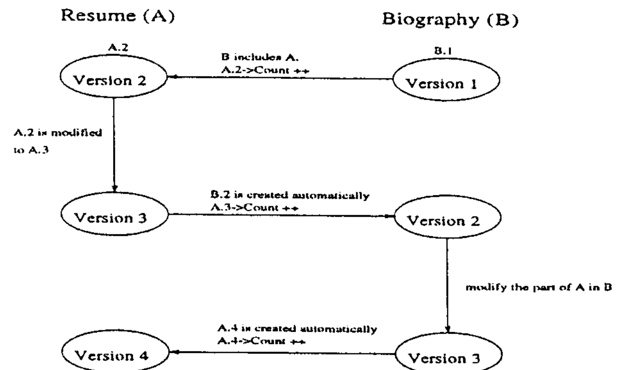

Figure 18: The relationship between the class Resume and the class Biography

4. CONCLUSION

In this paper, we have designed and implemented the query language for version control of the QBOE multimedia database system. We have considered all the implementation issues in version control: when to create a new version, how to represent the version, which versions in a database represent a consistent configuration of objects, and how a version reference, either static or dynamic, is executed.

References

[1] Y. I. Chang, S. H. Jair and H. N. Chen, "Design and Implementation of the QBOE Query Language for Multimedia Database Systems," *Proceedings of NSC, Part A: Physical Science and Engineering,* Vol. 21, No. 3, pp. 205-221, May 1997.

[2] W. I. Grosky, "Multimedia Information Systems," *IEEE Multimedia,* pp. 12-24, Spring 1994.

[3] D. Woelk, W. Kim, W. Luther, "An Object-Oriented Approach to Multimedia Databases," *Proc. of the ACM SIGMOD,* pp. 311-325, 1986.

[4] D. Woelk, W. Kim, "Multimedia Information Management in an Object-Oriented Databases System," *Proc. of 13th VLDB Conf.,* pp. 319-329, 1987.

[5] K. R. Dittrich and R. A. Lorie, "Version Support for Engineering Database Systems," *IEEE Trans. on Software Eng.,* Vol. 14, No. 4, pp. 429-437, April 1988.

[6] R. H. Katz, "Toward a Unified Framework for Version Modeling in Engineering Databases," *ACM Comp. Survey,* Vol. 22, No. 4, pp. 375-408, Dec., 1990.

[7] H. N. Chen, "Design and Implementation of Version Control for Multimedia Database Systems," *Master Thesis,* Dept. of Applied Mathematics, National Sun Yat-Sen Univ., Taiwan, R.O.C., June 1996.