Appendix A

A Design of MPEG-7 Testbed

This work [31] was published in Jannuary 2003. When it was initially developed (around year 2000), the Java platform still lacked several important tools such as the *collection framework*, and there were no (free) mature *persistence mapping* packages available. As the technology progresses, there are more tools available today. However, in the course of our research, this platform helps us to write CBIR related testing programs, and finally leads to the design of the scheme in Chapter 3. In the following sections, we briefly describe the design of this platform.

A.1 Motivations

The MPEG-7 XM [51] is a collective software set from the standard. Although it is the official and probably the most complete tool to develop new descriptions (Ds) and description schemes (DSs), it is mainly organized to demonstrate the extraction/matching results of using individual Ds (and/or DSs) and it has no database support. The database support is essential for a typical multimedia database application[52]. For research purposes, we need a platform that not only produces algorithm outputs, but also helps us analyzing the performance of an algorithm.

For new description development, what we care is how useful the description

is, and how well the extraction/matching algorithm performs. It is helpful if we have an environment that allows us to manipulate data structures and algorithms easily. For a new application development, a carefully arranged processing sequence together with dedicated database schema design may greatly enhance the overall performance. We also need a versatile interface for handling the inputs and outputs of MPEG-7 streams.

A.2 System Architecture

After analyzing the requirements and the use-cases, we have an overall architecture design. The system contains a number of concrete components, including descriptors, algorithms, and viewers, as shown in Fig. A.1. To manage those components, the platform has a component management unit that deals with the registration and association of the components. The database support is handled by the persistence management unit. These two units are not directly related to the descriptor design, but are necessary to the application and prototype development. Hence, they are utility units of the testbed. The most important part, though not explicitly mentioned in the requirements, is the framework. The framework is the programming styles and rules when we use the platform to develop new applications. It glues the scattered components and utilities by programming interfaces and class libraries.

The interfaces exposed to the designer and the developer are the framework and the implemented components. Since these two parts are designed to be system independent, this framework enables users to create or use the system components without knowing the underlying system configuration. Another feature of this architecture is component re-use. As projects (applications) are gradually developed on this platform, more and more components are registered on the system. It is beneficial to be able to re-use the previously developed modules.

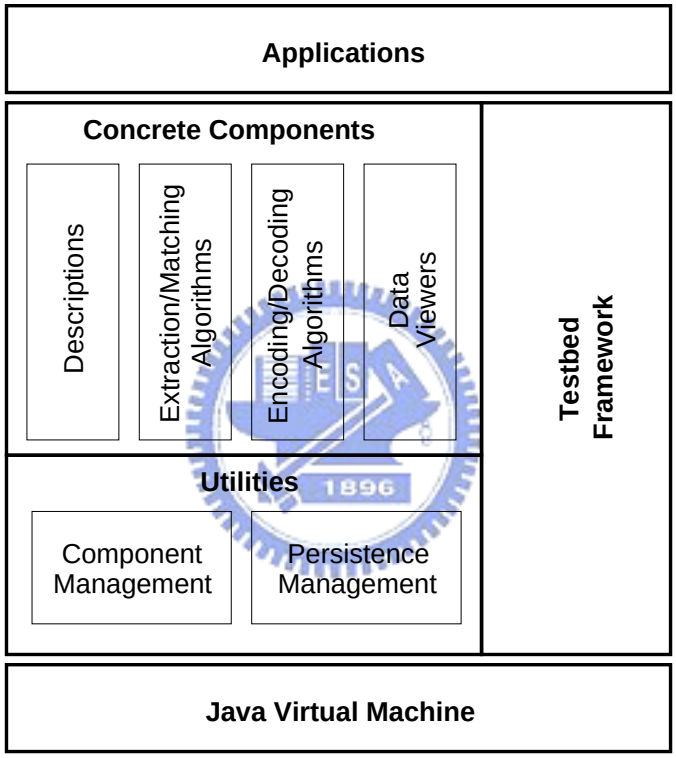

Figure A.1: Architecture of the software platform.

A.3 Framework Implementation

The framework is the most important part of our platform. For users, it is the interface that connects their applications to the testbed. For the system itself, it is the internal structure that holds the relationships among components. We identify the fundamental classes in our system as shown in Fig. A.2.

The Viewer/Data/DataAlg classes are the root of all subsequent components. Data, as the name implies, is a collection of bytes that represent the multimedia content. Specifically, it can be a picture, a video clip, a duration of sound, or any meaningful media data. The DataAlg is the algorithms associated with the data. For example, JPEG encoder and decoder are two DataAlgs bounded to JPEG data. Viewer classes are the renderers for Data classes, such as an image viewer, a sound player, and etc.

There are two special branches of DataAlg: one is for media data and the other is for meta-data (descriptions). The reason we separate these two types of algorithms is that the operations for media data and meta-data are significantly different. The operations applied to media data are encoding and decoding, while the meta-data are usually connected to feature extraction and feature matching. It is often reasonable to group encoder/decoder and extraction/matching algorithms in pairs. Note that we treat an MPEG-7 descriptor as a kind of media data, and hence the MPEG-7 bit-stream is the encoded result of a plain data structure. The associations among viewers and the data are not hard-coded, because a viewer can display more than one kind of data. For example, a vector viewer can display the histogram or the zig-zag scanned DCT coefficients of an image block. The association between data and algorithm is not one-to-one, because several algorithms may be associated with the same data.

The domain classes are the conceptual building structures of the platform. In constructing the testbed, we need to define more detailed behaviors of the classes. The Data interface is a wrapper of the real data, and it is the most generic type in

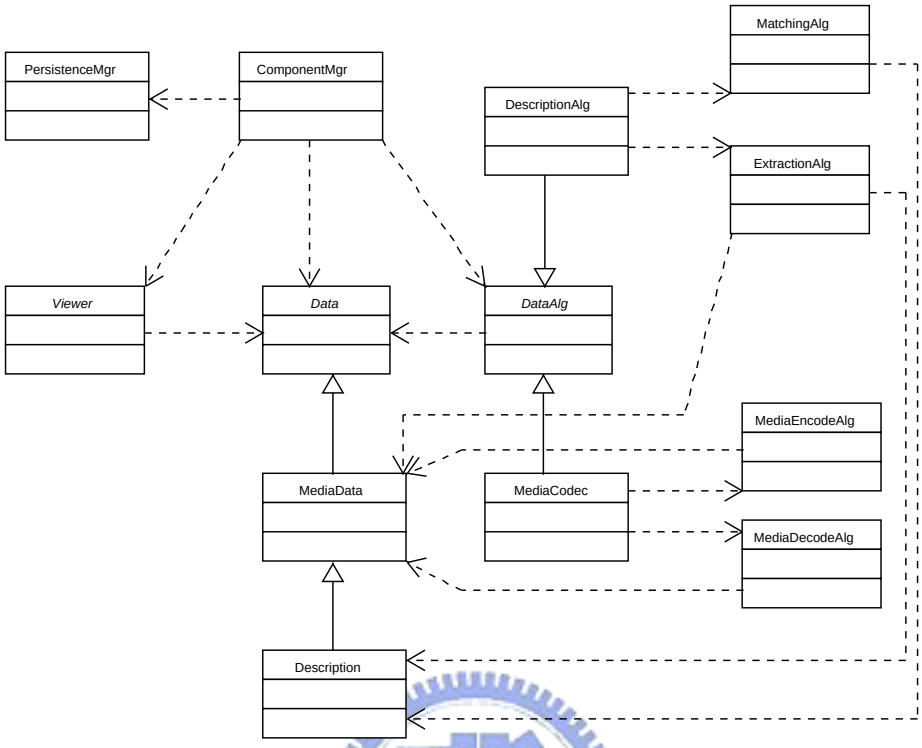

Figure A.2: MPEG-7 testbed domain classes.

the testbed. It acts as a tag interface indicating if the wrapped object is acceptable by this system. Programmers can retrieve MIME-type information through this interface if the system can recognize the object when it is constructed.

The DataAlg is also a tag interface, which represents the algorithms associated with the Data object. It is the root of all algorithm classes. It intends to serve as a common interface of the system to manage algorithms. For the developers (sometimes the designers), it is not directly used as often as its inherited classes. For example, the encoding/decoding algorithms consume MediaData. We use MediaCodec to group them for management, and we use the inherited MediaEncodeAlg class to implement the encoder.

A Viewer is used to view the data object. Following the notations of the model-view-controller (MVC) pattern[53, pp.293-303], the Viewer interface merely indicates that it would co-operate with the model object (the Data) and it provides

the general application programming interface (API) for feeding the data into the viewer. Note that we do not define the controller interface in our system, because different applications demand different interactions between a view object and a model object.

A.4 Applications

Based on the testbed, we have developed several applications for testing, simulation, and demonstration. In this section, we give a few application scree-shots in Fig. A.3.

At the first milestone, the framework was designed and a few descriptors were implemented. The viewer components and the utility unit were almost empty at that time. In order to demonstrate the feature matching, we wrote a *servlet* to parse the query file name from the URL and to present the matched images as thumbnails (Fig. A.3(a)) on a web page. Since we did not have the persistence mechanism, all the feature extraction/matching (including the images in the "database") were done at run-time. Then, several viewers were implemented, and we could present the extraction in a self-contained *applet* as shown in Fig. A.3(b). A few months later, the first version of the utility unit was complete, and we could demonstrate all the functionality in a full-fledged application. Figure A.3(c) shows feature extraction and image/feature viewing; Figure A.3(d) shows the single-descriptor matching functionality; Figure A.3(e) shows a weighted multi-descriptor matching scheme; and Figure A.3(f) shows a multi-stage weighted multi-descriptor matching scheme.

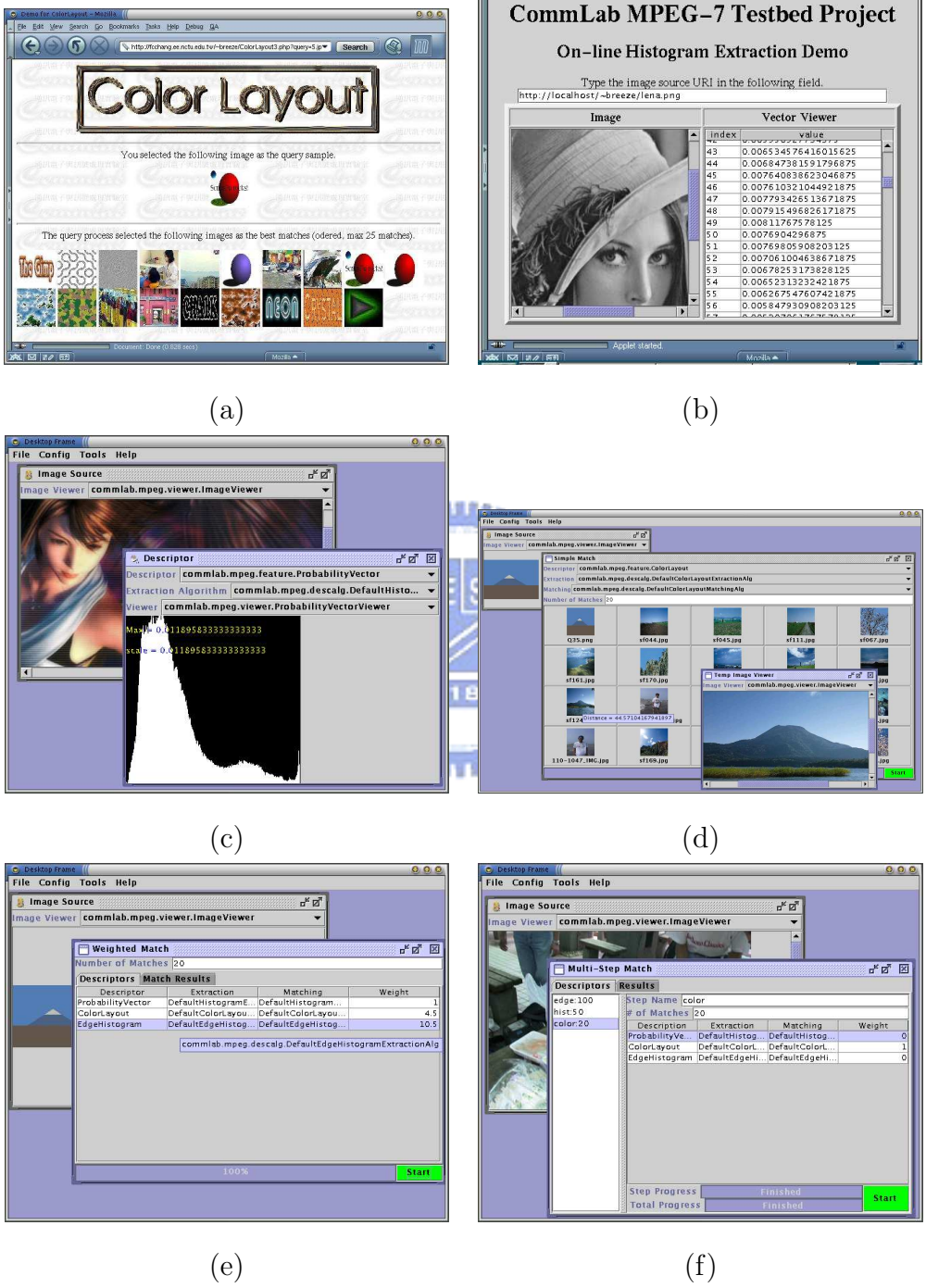

Figure A.3: Screen-shots of applications.

Appendix B

IPMPX Implementation

The intellectual property management and protection (IPMP) was first introduced in the MPEG-4 [54] standard to be a DRM mechanism in the MPEG world. To be distinguished, the first version is called *IPMP hook* [55], and the evolved version is called *IPMP Extension (IPMPX)* [56]. There were several related efforts which incorporate IPMPX into systems. One of the project is the *MPEG-21 Multimedia Testbed for Resource Delivery* [57]. We found that the IPMPX subsystem should be re-implemented on the Testbed, because the implementation in the IM1 reference software is tightly coupled with its underlying infrastructure. In the following sections, we describe the design and implementation of IPMPX for the Testbed.

B.1 IPMPX Overview

The fundamental concept of the MPEG-4 IPMPX is *virtual terminal*. It is a subsystem of an MPEG-4 terminal. All the tools inside the virtual terminal communicate to each other (including the terminal) by *messages*. Below we briefly describe the major components of IPMX.

Message Router

Since the communications among tools and the terminal are by means of messages, there should be a mechanism to deliver message from one to other. The Message Router (MR) is a conceptual component to represent the mechanism. The format of a message has at least two fields: source and destination *context ID*. A context ID is a unique number within an MPEG-4 terminal to identify a certain component. From the viewpoint of a virtual terminal, the physical terminal is one of the components it can communicates to. Thus, the context ID zero is reserved to indicate the physical terminal. According to the context IDs, the MR can deliver messages within the virtual terminal.

Tool Manager

The Tool Manager (TM), as its name suggests, is the component to manage IPMP Tools. It is also a conceptual component to represent the mechanism to resolve, obtain (including download), instantiate, and connect/disconnect a tool. When it receives the IPMP Tool List Descriptor, it tries to resolve all the tool IDs by the information specified in the descriptor. The next step is to check whether a tool is locally available or not, and download the missing tools through the *tool manager interface*. Then, it instantiates the tool when the corresponding Tool Descriptor Pointer is received, and initializes the tool by the information specified in the IPMP Tool Descriptor. According to the tool descriptor, a tool would be connected to a certain Control Point if the control point specifier is not null.

IPMP Tool

An IPMP Tool is responsible to perform a certain operation which may be a full or part of a protection functionality. Some IPMP Tools directly process the content stream, and some tools are not. The former category of tools should be connected to a control point to be able to receive the content stream. The latter category

of tools, though only process messages, are also important to provide service-like functions. For example, a license verification tool does not process audio/video streams directly, but it is used to enable/disable the content-processing tools.

Control Point

A Control Point is a filter which resides somewhere along the content processing path. Actually it is a container to connect a series of content processing tools. That is, the content stream is processed by the contained IPMP Tools one after another.

B.2 Design and Implementation

Due to some historical reasons, the IPMPX functionality was not considered and verified in depth at the first Testbed design stage. To integrate IPMPX into the Testbed without too much burden, we decide to abstract the Testbed as a stub component in the design. All the Testbed related functions are aggregated in the class *Terminal*, including the funcion of parsing IPMP descriptors from the initial object descriptor and from the elementary stream descriptors.

Another issue is the context ID. A context ID should be unique inside the terminal, and it is also used for searching a tool instance. We decide to manage the IDs in a tree structure. We design a class *IPMPContext* to hold all context IDs and instantiated IPMP Tools. The top most context contains the references to the MR, TM, and Terminal for easy access to these components. Figure B.1 shows the relationships of various context types. The abstract class IPMPContext defines that a context may have a parent context and may have none or many child contexts. The *IPMPTopContext* is the top-most context and is designed as a singleton. The *IPMPObjContext* is the object level context which is a placeholder in the Testbed. The *IPMPESContext* is the elementary stream level context, which may contain several *IPMPToolContex*s. All the other components are related to each other through the context tree.

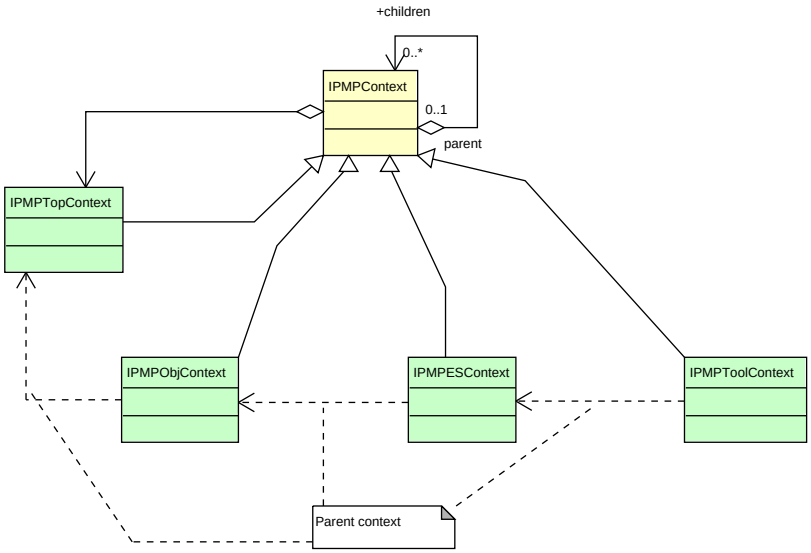

Figure B.1: The relationships of various context types.

The above is the initial design of the IPMPX subsystem. The final integration design [58] is shown in Fig. B.2. The gray boxed are the Testbed components which are affected when integrating IPMPX subsystem into the first version architecture. This design was carried out and completed by Chen-Wei Fan. A more detailed report can be found in [58][59]

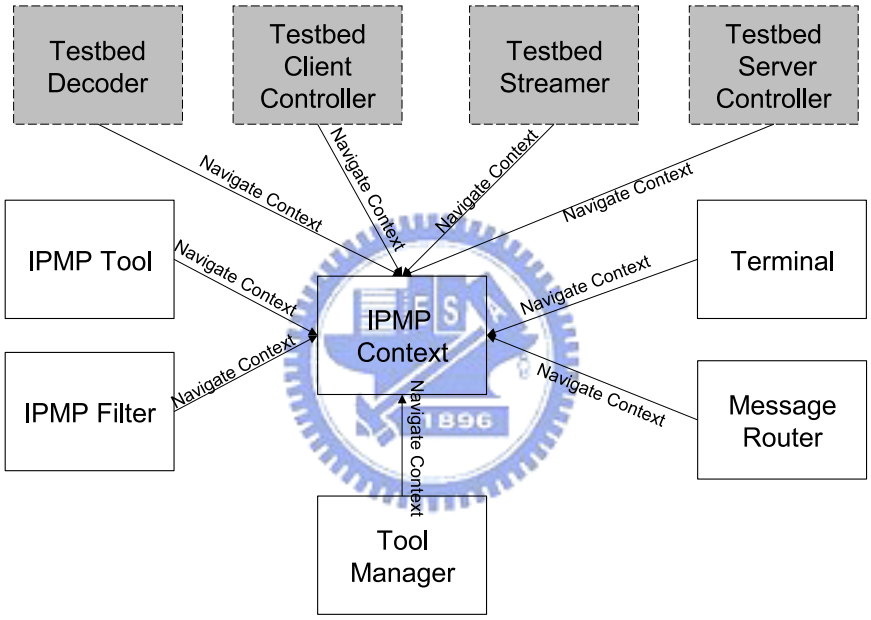

Figure B.2: The relationships of IPMPX components and the Testbed.