Chapter 16

Abstractions for Programming

Most of the current commercially available multimedia applications are implemented in procedure-oriented programming languages (structural languages) such as C. In the past, multimedia-specific functions (e.g., changing the volume while playing an audio passage) were called, and respectively controlled, through hardware-specific libraries or device drivers.

Unfortunately, the application code of most commercial multimedia application programs are still highly dependent on hardware. The exchange of a multimedia device still often requires a re-implementation of important parts of the application program. This also happens when devices produced by different companies with similar or identical functionalities are exchanged. Consider, for example, video cards. Cards with the same functionalities are produced by companies like Apple, IBM, Parallax and RasterOps. However, the functions for accessing the cards are completely different. With the advent of common operating system extensions, this problem is attacked.

Some applications are implemented with the help of *tools*. These tools either directly generate the code or manage routines which can be used by the application in order to integrate the device units into the application. When these devices are exchanged, these applications often require either generation of new code or the tool must be changed and new interaction methods with the device units are needed.

A comparison can be made to the technique of programming with floating point numbers. Different computers, which support floating point numbers, differ in their architectures, instructions and interfaces. Sometimes, RISC architectures or parallel processors are used. Despite the variety of architectures, only a few standard formats, such as the IEEE format, are used for the presentation of numbers. The programmers mostly use the so-called *built-in functions* of higher programming languages to make use of floating point processing capabilities.

In contrast to multimedia environments, well-defined abstractions in higher programming languages can be found in the form of *data types* (e.g., float type in C). This approach hides the actual hardware configuration from the application without any major loss of performance.

In research, object-oriented approaches to the programming of multimedia systems and especially applications, have been used [Bla91a, GBD+91, RBCD91, FT88, LG90a, SHRS90, SM92a]. Also, interfaces to communication systems are often implemented via object-oriented approaches.

Multimedia objects allow a fast integration in their environment despite their different capabilities, properties and functions. Development of a separate language or extensions of a compiler are not necessary. Multimedia can become an integral part of the programming environment if the proper class hierarchy is used. Unfortunately, the various currently used class hierarchies are very different. There is no generally accepted optimal class hierarchy.

This chapter describes different programming possibilities for accessing, and respectively representing multimedia data. Further, this chapter gives an overview of abstraction levels such as *libraries*, *system software*, *higher procedural programming languages* and *object-oriented approaches* going into more detail in some of the approaches.

16.1 Abstraction Levels

Abstraction levels in programming define different approaches with a varying degree of detail for representing, accessing and manipulating data. We describe in this

chapter the abstraction levels with respect to multimedia data and their relations among each other (shown in Figure 16.1). A multimedia application may access

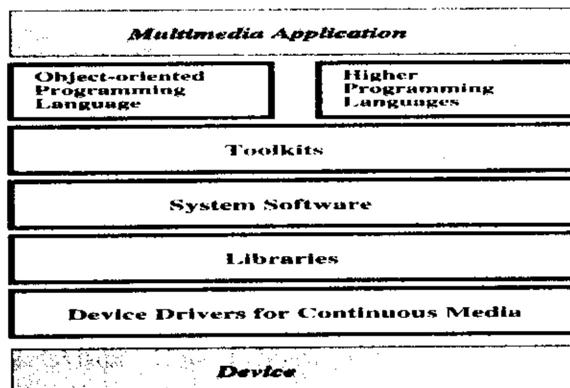

Figure 16.1: *Abstraction levels of the programming of multimedia systems.*

each level.

A *device* for processing continuous media can exist as a separate component in a computer. In this case, a device is not part of the operating system, but is directly accessible to every component and application. A *library*, the simplest abstraction level, includes the necessary functions for controlling the corresponding hardware with specific device access operations.

As with any device, multimedia devices can be bound through a device driver, respectively the operating system. Hence, the processing of the continuous data becomes part of the *system software*. This requires several properties described in Chapter 9 from the multimedia operating system, for example, appropriate schedulers, such as rate monotonic scheduler or earliest-deadline-first scheduler. Multimedia device drivers embedded in operating systems simplify considerably the implementation of device access and scheduling. For example, an operating system could resolve the register allocation for individual devices so that no collision occurs. In the case where allocation of registers for some devices (e.g., ATM host interface) is controlled through the Operating System (OS) and for other devices through applications (e.g., video card), the application programmer must be careful when assigning the registers.

Dedicated programming languages, such as programming for *Digital Signal Processing* (DSP), allow for the implementation of real-time programs. The corresponding program mostly runs in a *Real-Time Environment* (RTE) separate from the actual application. It is not very common today that an application software is programmed in the RTE.

Higher procedural programming languages build the next abstraction level. They are the languages most often used to implement commercial multimedia applications. Further, they can contain abstractions of multimedia data [SF92]. The code generated from the compiler can be processed through libraries, as well as through a system interface for continuous data.

More flexibility for the programmer is provided via the abstraction level – *an object-oriented environment.* This environment provides the application with a class hierarchy for the manipulation of multimedia. Also in this case, the generated or interpreted code can be processed and controlled through libraries, as well as through a system interface for continuous media (see Figure 16.1).

16.2 Libraries

The processing of continuous media is based on a set of functions which are embedded into libraries. This is the usual solution for programming multimedia data. These libraries are provided together with the corresponding hardware.

The device driver and/or library, which controls all available functions, also supports each device. (In the early DiME project at IBM Heidelberg, for example, a large number of different audio and video components were supported by corresponding hardware cards which were either connected directly to the workstation or were implemented as extension cards [SSSW89].) Here, the libraries differ very much in their degree of abstraction. Some libraries can be considered as extensions of the graphical user interface, whereas other libraries consist of control instructions passed as control blocks to the corresponding driver. Consider, for example, some functions supporting the IBM's early *Audio Visual Connection (AVC)*:

```
acb.channel = AAPI_CHNA
```

```
acb.mode = AAPI_PLAY
...
aud_init(&acb)  /* acb is the audio control block */
...
audrc = fab_open(AudioFullFileName,AAFB_OPEN,AAFB_EXNO, 0,&fab,0,0,0,0);
fork(START IN PARALLEL)
aud_strt(&acb)
displayPosition(RelativeStarttime, Duration)
...
acb.masvol = (unsigned char) Volume
audrc = aud_crtl(&cb)
...
```

Libraries are very useful at the operating system level, but there is no agreement (and may never be) over which functions are best for different drivers, i.e., which functions should be supported. As long as neither sufficient support of operating systems for continuous data nor further integration into the programming environment exist, there will always be a variety of interfaces and hence, a set of different libraries.

16.3 System Software

Instead of implementing access to multimedia devices through individual libraries, the device access can become part of the operating system. An example of access to multimedia devices and support for continuous media processing implemented in operating system is the experimental *Nemo* system from the University of Cambridge [Hyd94] (shown in Figure 16.2). The Nemo system consists of the Nemo Trusted Supervisor Call, running in supervisor mode, and three domains running in user mode: *system*, *device driver* and *application*.

The *Nemo Trusted Supervisor Call (NTSC)* code implements those functions which are required by user mode processes. It provides support for three types of processes. *System processes* implement the majority of the services provided by the operating system. *Device driver processes* are similar to system processes, but are

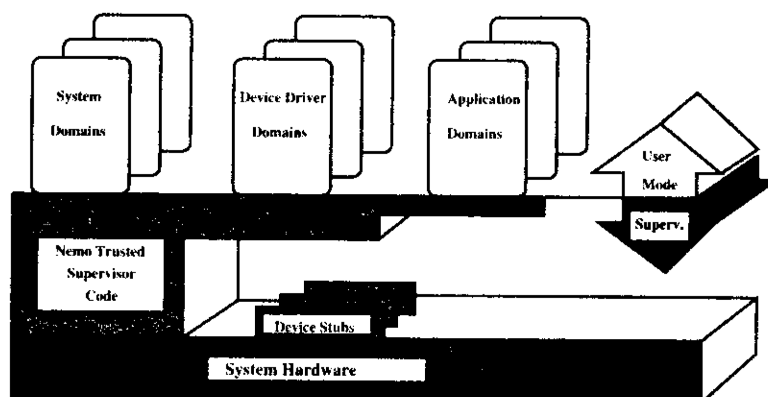

Figure 16.2: *Structure of the Nemo system.*

distinguished by the fact that they are attached to device interrupt stubs which execute in supervisor mode. Application processes contain user programs. Processes interact with each other via the system abstraction – InterProcess Communication (IPC) – which is implemented using low-level system abstractions ***events*** and, if required, ***shared memory***. These system abstractions support the continuous media communication among processes [Hyd94].

The NTSC calls are separated into two classes, one containing calls which may only be executed by a suitable privileged system process such as kernel, the other containing calls which may be executed by any process. Further, NTSC is responsible for providing an interface between a multimedia hardware device and its associated driver process. This device driver implementation ensures that if a device has only a low-level hardware interface to the system software, code can be implemented within a device driver stub to implement a higher level interface. This allows the system builder to trade off hardware complexity and cost against the processor cycles required to implement a high-level device interface.

16.3.1 Data as Time Capsules

Some special abstractions, such as the *time capsule* [Her90], are seen in multimedia systems as being related to files systems. These file extensions serve as storage, modification and access for continuous media. Each *Logical Data Unit (LDU)* carries

in its time capsule, in addition to its data type and actual value, its valid life span.

This concept is more useful for video than for audio. For example, if a video has 25 frames per second, each frame has a valid life span of 40 ms. The read access during a normal presentation occurs at this rate. The presentation rate changes for fast forward, slow forward or fast rewind processes. This can be achieved as follows:

1. The presentation life span of a data unit (e.g., a video frame) can change. For example, in a slow forward process, each frame is valid for a longer period of time.

2. The valid life span is not considered. Instead, the choice of the LDUs, which are specified through the time capsule, is influenced. In the fast forward process, some data units are skipped, but the valid life span for each frame is not changed. In the slow forward process, the presentation of individual frames occurs twice or several times. Note that this simple implementation is possible with uncompressed data. In the case of compressed motion pictures (e.g., MPEG), the information change of the consecutive frames needs to be considered and therefore an arbitrary individual frame cannot be skipped. However, access to I-frames can only be implemented.

The *time capsule* work can be extended by the refinement of LDUs (e.g., pixel, image frame, video sequence, etc.). The modification of the data rate should not follow for each sample value, but for a sequence of sample values. At the video output device, each frame must have the same valid life span because the modification of the physical rate is not possible. A similar concept, connection of data with rate, is presented in [GBD+91].

16.3.2 Data as Streams

A well-known, used and implemented abstraction at the system level is the *stream*. A stream denotes the continuous flow of audio and video data. Prior to this flow, the stream is established between source(s) and sink(s). This is equivalent to the setup of a connection in a networked environment. Subsequently, operations on a *stream* can be performed such as play, fast forward, rewind and stop.

In Microsoft Windows, a *Media Control Interface (MCI)* provides the interface for processing multimedia data [Mic91b, Mic91c]. It allows the access to continuous media streams and their corresponding devices.

For further considerations of streams and any kind of operating system requirements and basic system support of abstractions for programming continuous media application (e.g., page locking, preemptive scheduling, system call timeouts, etc.), Chapter 9 provides sufficient guidance.

16.4 Toolkits

A simpler approach (from the user perspective) in a programming environment than the system software interface for control of the audio and video data processing can be taken by using *toolkits* [AGH90, AC91]. These toolkits are used to:

- Abstract from the actual physical layer (it is also done in a limited way by the libraries).
- Allow a uniform interface for communication with all different devices of continuous media (with eventual input of quality of service parameters).
- Introduce the client-server paradigm (here, the communication can be hidden from the application in an elegant way).

Toolkits can also hide process-structures. It would be of great value for the development of multimedia application software to have the same toolkit on different system platforms, but according to current experiences, this remains to be a wish, and it would cause a decrease in performance.

Toolkits should represent interfaces at the system software level. In this case, it is possible to embed them into the programming languages or object-oriented environment. Hence, we describe the available abstraction in the subsequent section on programming languages and object-oriented approaches.

16.5 Higher Programming Languages

In the following, procedural higher programming languages will be called ***High-Level Languages (HLL)***. In such an HLL, the processing of continuous media data is influenced by a group of similar constructed functions. These calls are mostly hardware- and driver-independent. Hence, their integration in HLLs leads to a wishful abstraction, supports a better programming style and increases the productivity. Programs must be capable of supporting and effectively manipulating multimedia data. Therefore, the programs in an HLL either directly access multimedia data structures, or communicate directly with the active processes in a real-time environment. The processing devices are controlled through corresponding device drivers. Compiler, linker and/or loader provide the required communication between the application program and the processing of continuous data. There does not yet exist a programming language which includes special constructs for the manipulation of multimedia data, besides possibly programming languages in the digital signal processing domain, which exist mostly at the assembler level to achieve the best time behavior of a program.

Media can be considered differently inside a programming language. In the following subsections, different developed variants are discussed. First results have been published in [SF92] and [Ste93b].

16.5.1 Media as Types

The following example shows the programming expression in an OCCAM-2 similar notation [Lim88, Ste88]. OCCAM-2 was derived from *Communication Sequential Processes (CSP)* [Hoa85]. This language is used for the programming of transputers [Whi90]. This notation was chosen in the following examples because of its simplicity and embedded expressions of parallel behavior. This does not mean that programming must be enforced this way or that this is a better way of processing multimedia data.

```
a,b REAL;
ldu.left1, ldu.left2, ldu.left_mixed AUDIO_LDU;
```

```
...
WHILE
   COBEGIN
      PROCESS_1
         input(micro1,ldu.left1)
      PROCESS_2
         input(micro2,ldu.left2)
   ldu.left\_mixed := a * ldu.left1 + b * ldu.left2;
   ...
END_WHILE
...
```

One of the alternatives to programming in an HLL with libraries is the concept of *media as types*. Here, the data types for video and audio are defined. In the case of text, character is the type (the smallest addressable element). A program can address such characters through functions and sometimes directly through operators. They can be copied, compared with other characters, deleted, created, read from a file or stored. Further, they can be displayed, be part of other data structures, etc. There is no plausible reason, known to the authors, why the same functionality cannot be applied to continuous media. The smallest unit can be the LDU. As described in Section 2.6, these data units can be of very different granularity (and therefore of different size and duration). In the above described example, two LDUs from microphones are read and mixed. The following example describes the merging of a text and motion picture. It is interpreted as the overlay of the text onto the motion picture:

```
subtitle TEXT_STRING;
mixed.video, ldu.video VIDEO_LDU;
...
WHILE
   COBEGIN
        PROCESS_1
             input(av_filehandle,ldu.video)
             IF new_video_scene
```

```
                input(subtitle_filehandle,subtitle)
            mixed.video := ldu.video + subtitle
        PROCESS_2
            output(video_window,mixed.video)
    ...
END_WHILE
...
```

An application for the merging example is a provision of subtitles in a video clip. For example, a distribution service can transmit a movie parallel with audio and subtitles in many languages. The user decides the combination. It is already done with stereo tone where two languages are partially provided. The mixture of two visual media, except the case of having a picture inside another picture, is not provided in this form. The mixture of text and video can already be implemented in a simple way through using, e.g., the teletext-decoder integrated in television devices. Note, that in the above described subtitle example, an implicit type conversion must occur. Variables of different types (VIDEO_LDU, TEXT_STRING) are added (ldu.video + subtitle) and at last again assigned to a variable (mixed.video) of one of these types (VIDEO_LDU). During the merge, respectively the adding process, their relative position and duration can be specified. Besides a standard value (e.g., center the subtitle in the lower part of the picture) specified a priori, this relative position can be defined freely by the programmer at the initialization phase. The duration is determined in the program through an explicit fade-in operation. It can also be defined relative to the scene duration at the initialization. Note that several possibilities for the duration specification are described in Chapter 15.

In this area, we gathered the following experiences:

- The real-time processing of LDUs with a very fine granularity is complicated in an HLL, respectively there are only conditionally predictable. An example of an LDU at very fine granularity is an individual audio sample. The following solutions are possible:
  - The HLL actually consists of two programming environments, which are mixed together in an application program. The real-time environment

contains digital signal processing algorithms, which are generated, modified, improved, etc. [RS78]. The non-real-time environment contains the whole conventional programming. The application programmer does not notice that he deals with those two environments. Both environments are concealed for the programmer. Hence, the code of both environments can be mixed.

- As an alternative, the algorithms of the corresponding environments can be contained in different functions or procedures. It is possible to call all real-time functions through a *Real_ Function*, and all remaining functions through a *Function*. Both kinds of functions can be used together. How far a *Call by Reference* of continuous data is possible, depends on the processing model of the continuous data, which is not easy to implement. If a dedicated memory space is handled by a special-purpose signal processor, then this approach becomes even more difficult to be implemented. Using a main processor in real-time mode, it can be implemented.
- A communication concept between the two programming environments can be introduced in an HLL. An application would exist of two separate modules where each module is defined for one programming environment. These modules, which consist of at least one process, would exchange the necessary information through the multimedia communication concept.

- An LDU with a gross granularity exists when, for example, a video or audio sequence is accessed only as a whole unit. In this case, the individual media elements cannot be accessed (e.g., beginning of the second scene).
- With respect to the *granularity of the LDU*, it is important to find the proper size of an LDU as the data type:

  For example, in the case of audio, the audio blocks (75 per second), known from CD technology, should be the accessible units. In the case of video, the minimal granularity should be the video frames (i.e., not image segments, lines, columns, pixels, etc.). Also, short video clips up to duration of two seconds can be defined as an LDU. This makes sense, for example, for compressed video, where besides an intraframe compression image, also images with difference values, interframe coded, are transmitted. A typical sequence contains two

intraframe coded images per second, the rest (28 images) are interframe coded. Hence, an LDU would have the duration of 0.5 seconds.

From a pragmatic point of view, a manipulation of pixels for a discrete cosine-transformation or a fast Fourier transformation should not be part of an HLL. This should further consist as part of the digital signal processing with the corresponding software and hardware. But the HLL should have access to the DSP algorithms as a whole.

- The meaning of the operators + *(addition)*, - *(removal)*, etc. is not only media-dependent, but also application-specific. The addition of two video images can mean an overlapping of two images (with transparent colors) or only a mixture of luminance values. Here, a consent for the general interpretation is necessary.

16.5.2 Media as Files

Another possibility of programming continuous media data is the consideration of continuous media streams as *files* instead of data types.

```
file_h1 = open(MICROPHONE_1,...)
file_h2 = open(MICROPHONE_2,...)
file_h3 = open(SPEAKER,...)
...
read(file_h1)
read(file_h2)
mix(file_3, file_h1,file_h2)
activate(file_h1, file_h2, file_h3)
...
deactivate(file_h1, file_h2, file_h3)
...
rc1 = close(file_h1)
rc2 = close(file_h2)
rc3 = close(file\_h3)
```

The example describes the merging of two audio streams. The physical file is associated during the open process of a file with a corresponding file name. The program receives a *file descriptor* through which the file is accessed. In this case, a device unit, which creates or processes continuous data streams, can be associated with a file name.

Read and *write functions* are based on continuous data stream behavior. Therefore, a new value is assigned continuously to a specific variable which is connected, for example, with one read function. On the other hand, the read and write functions of discrete data occur in separate steps. For each assignment of a new value from a file to the corresponding variable, the read function is called again.

In a *seek function*, the pointer in a file can be positioned to particular places which correspond to the beginning of an LDU. Continuous data are also often played from a source of non-persistent data. A microphone and camera are examples of such sources. For these files, a seek function cannot be performed. This can be compared with the reception of discrete data from a keyboard. This kind of file processing is widespread in the UNIX™environment. Here, the most device units are handled at their interfaces to the applications as files (either as a *stream device* or as a *block device*). The programming of devices must be extended corresponding to Leungs *active devices* [LLM+88]. All file-similar functions can be used, additionally it is also possible to activate and deactivate a device. An *activate function* means that the actual data transmission starts and a *deactivate function* means that the transmission stops.

Using this kind of programming of continuous data, the number and functionality of the operations with continuous data (in comparison to programming with media data types) is limited. This approach can be seen as the programming of data streams.

16.5.3 Media as Processes

The processing of continuous data contains a time-dependency because the life span of a process equals to the life span of a connection(s) between source(s) and destination(s). A connection can exist locally, as well as remotely. Under this consideration,

it is possible to map continuous media to processes and to integrate them in an HLL.

```
     PROCESS cont_process_a;
     ...
     On_message_do
         set_volume ...
         set_loudness ...
         ...
     ...
   [main].
pid = create(cont_process_a)
send(pid, set_volume, 3)
send(pid, set_loudness)
...
```

In the above example, the process *cont_process_a* implements a set of *actions (functions)* which apply to a continuous data stream, Two of them are the modification of the volume *set_volume* and the process of setting a volume, dependent from a band filter, *set_loudness*.

During the creation of the process, the identification and reservation of the used physical device(s) occur. The different actions of the continuous process are controlled through an IPC mechanism. For example, the transmission of continuous data is controlled by sending signals and messages. The continuous media process determines itself how the accessed actions are performed.

Thus, the processing can be done either once or continuously, meaning that during the entire transmission of continuous data:

- The loudness is determined once by a device driver call. The driver loads a certain storage content which is used by the running process controlling an audio board.
- If the main processor passes the audio data further from a file to the communication system, then the loudness can be changed here. Thus, the compression

and coding must be considered. In this example, it is assumed that an uncompressed PCM-coded audio signal with 64 Kbits/s is present. The continuous process transmits these data and changes, as well as the loudness, according to the desired value.

The present variants for the integration of a multimedia programming in an HLL require the properties of the programming language described in the section below.

16.5.4 Programming Language Requirements

The processing of continuous data is:

- Controlled by the HLL through pure asynchronous instructions (typically, through the use of a library).
- An integral part of a program through the identification of the media, respectively data streams with data types, variables, files or processes.

Therefore, the HLL should support a *parallel processing* as was presented in all examples of HLL programming of continuous media. Thus, it is of secondary importance if the number of processes is known at compile time or if it is defined dynamically at run-time.

Interprocess Communication Mechanism

Different processes must be able to communicate through an Inter-Process Communication mechanism (IPC). This IPC mechanism must be able to transmit audio and video in a timely fashion because these media have a limited life span. Therefore, the IPC must be able to:

- Understand a priori and/or implicitly specified time requirements. These requirements can be specified using QoS parameters or they can be extracted from the data type (if a medium is implemented as a data type).

- Transmit the continuous data according to the requirements.
- Initiate the processing of the received continuous process on time.

The generated heap from the compiler is limited in its size, the location and properties are determined by the compiler. The processing of time-critical data requires a careful assignment and manipulation of the storage space. The IPC and communication between different programs must happen effectively. The performance analysis in the multimedia and high-speed communication systems show that the most time-consuming operation is the *copying of data* operation. System-wide uniform *buffer management* at the system software layer extracts this problem. A virtual copying means that the access rights onto the buffer spaces are passed to other components. For example, this approach was implemented in the multimedia communication system HeiTS [HHS91]. The HLL compiler for continuous media must use this buffer management. The same is true for the IPC implemented in this language.

Audio and video *processes* require the availability of *real-time* processing. This can be implemented, as described in Section 16.4.1, by combining two programming environments. The HLL should support a clear data type specification.

Language

The authors see no demand for the development of a new dedicated language. A partial language replacement is also quite difficult because cooperation between the real-time environment and the remaining programs requires semantic changes in the programming languages. The IPC must be designed and implemented in real-time, the current IPC can be omitted.

A language extension is the solution proposed here. For the purpose of simplicity, in the first step, a simple language should be developed which satisfies most of the above described requirements. An example of such a language is OCCAM-2. Some real-time systems are implemented in this parallel programming language today. An alternative is a parallel C-variant for the transputer. In the long run, ADA still provides a good concept as a language basis.

16.6 Object-oriented Approaches

The object-oriented approach was first introduced as a method for the reduction of complexity in the software development and it is used mainly with this goal today. Further, the reuse of software components is a main advantage of this paradigm. The basic ideas of object-oriented programming are: *data encapsulation* and *inheritance,* in connection with *class* and *object* definitions. The programs are implemented, instead of using functions and data structures, by using classes, objects, and methods.

Abstract Type Definition

The definition of data types through abstract interfaces is called *abstract type definitions.* The abstract type definition is understood as an interface specification without a knowledge and implementation of internal algorithms. This data abstraction hides the used algorithm.

In a distributed multimedia system, abstract data types are assumed for virtual and real device units such as cameras and monitors. For example, an interface, which contains a function *zoom*, can also contain a parameter which specifies the actual position in an area from 10 ... 500. However, this specification does not describe the actual implementation.

Class

The implementation of abstract data types is done through *classes.* A class specification includes an interface provided to the outside world.

For example, in a class *professional_camera*, the operations *zoom* and *set_back_light* are defined and implemented. If the objects, which represent a closed class, use only relative position entries, the implementation of the *zoom* operation needs to transform the absolute values into the necessary relative parameters.

Object

An *object* is the instance of the class. Therefore, all objects, derived from the same class include the same operations as an interface to the outside world. An object is created at run-time of the system. It includes a set of operations, which are called *methods*. Additionally, each object has an *internal state*, which exists during the life span of the object, but it can only be accessed using the methods associated with this object. It can be compared with a global variable assigned to a process, but not with local variables of functions and procedures (as is implemented in most programming languages). Objects communicate among each other through the exchange of messages. Thus, a *message* calls the corresponding method of the target object.

In a distributed multimedia environment, virtual units are considered to be objects. Thus, corresponding methods represent operations on the devices. The method *play* of a VCR object (*Video Cassette Recorder*) is mapped to the *play_operation* of the corresponding VCR device driver. Multimedia data units (the LDU's images, audio and video clips) can also be considered objects.

Inheritance

One of the most important properties of object-oriented systems is *inheritance*. Classes contain, besides the root and leaves of the hierarchy, superclasses and subclasses (fathers and sons).

For example, let the class *professional_camera* be a subclass of the class *camera*. Methods such as *autofocus_on* and *focus* are defined in the class *camera*. The *professional_camera* class also has the method *zoom*. An object, which is derived from the *professional_camera*, can use the method *zoom*, as well as the operations *focus* and *autofocus_on*.

The main problem has been and remains to be the design of a clear and uniform class hierarchy for a multimedia system.

Until now, only simple inheritance was considered. For example, for the application

interface of a conference application, it is not possible to explicitly combine all necessary devices for each conference. Such an application would be dependent on a certain set of device types which are bound together in the required configuration. Device binding often includes different basic devices. A conference object inherits properties of different objects in an object-oriented environment, therefore a multi-inheritance is often useful.

Another type of inheritance is used by the consideration of interfaces. The same interface is provided for different classes, but the implementations can be very different. So, the operator + *(addition)* could contain, according to the used data type, slightly different semantics and implementation. The addition of audio LDUs means the mixing of audio signals. Additionally, when applied to two video data streams, it can mean, for example, simultaneous presentation of both information streams in different halves of a window.

Polymorphism

Polymorphism is related to the property of inheritance indicating when the same name of a method is defined in several classes (and objects) with different implementations and functionalities. For example, the function *play* is used with audio and video data. It uses different device units for each medium. The data can come either from a file of a local file environment or from an audio-video sequence of an external device. Inside of the object-oriented approach, for example, *play* is defined in different classes. According to which object must perform the operation, the corresponding method is chosen.

This concept is especially useful with respect to system use because the complexity of different types and device units is reduced and there is a common set of method names for classes and objects of different media. On the other hand, polymorphism can also very easily cause programming errors that are difficult to find. Hence, this abstraction strongly complicates the implementation. This can occur easily through unwanted, multiple identical method names. In an object-oriented environment, according to Wegener's definition [Weg87, Nie89], multimedia programming is achieved through the implementation and extension of class hierarchies.

The following sections describe different class hierarchies that support multimedia systems and applications. The examples and actual implementations were developed in the language C++, but the results are independent of a specific language [SF92]. In the authors' opinion, there will be many different class hierarchies in the future and they will be connected through complex relations to provide required interactions. However, the resulting complexity will not be easy to handle.

16.6.1 Application-specific Metaphors as Classes

An application-specific class hierarchy introduces abstractions specifically designed for a particular application. Thus, it is not necessary to consider other class hierarchies. This approach leads to a number of different class hierarchies. Furthermore, using this approach, one very easily abandons the actual advantage of object-oriented programming, i.e., the reuse of existing code.

Unfortunately, this is currently the most used solution, which has led to different kinds of class hierarchies. Although, for similar applications, similar class hierarchies can be implemented. Therefore, a catalog of similar applications is necessary to use the existing knowledge for the development of a new application.

16.6.2 Application-generic Metaphors as Classes

Another approach is to combine similar functionalities of all applications. These properties or functions, which occur repeatedly, can be defined and implemented as classes for all applications. An application is defined only through a binding of this class. For example, basic functions or functional units can create classes. The methods of these classes inherit the general methods through integration of application-specific subclasses. In theory this approach sounds easy to follow. In practice, we have not yet a very useful set of basic/generic application classes looks like, because known implementations of application-generic classes only work well for a very restricted set of applications.

16.6.3 Devices as Classes

In this section we consider objects which reflect a physical view of the multimedia system. The devices are assigned to objects which represent their behavior and interface.

Methods with similar semantics, which interact with different devices, should be defined in a device-independent manner. The considered methods use internally, for example, methods like *start*, *stop* and *seek*. Some units can manipulate several media together. A computer-controlled VCR or a Laser Disc Player (LDP) are storage units which, by themselves, integrate (bind) video and audio. In a multimedia system, abstract device definitions can be provided, e.g., camera and monitor. We did not say anything until now about the actual implementation. The results show that defining a general and valid interface for several similar audio and video units, as well as input and output units, is quite a difficult design process. This is also demonstrated by the following abbreviated C++ program [SF92]:

```
class media_device
       {char *name;
        public:
                void on(), off();
       }; /* end media_device */

class media_in_device :
public media_device
       {private:
                 DATA data;
        public:
                 refDATA get_data();
       }; /* end media_in_device */

class media_out_device :
       {public:
                void put_data(refDATA dat);
       }; /* end media_out_device */
```

```
class answering_machine:
public media_device
       {private:
           list my_list; //  class for ADT list
           media_in_device recorder;
           media_out_device message_for_caller, message_from_caller;
           RefDATA information; //  text a caller hears
           void display_position();
        public:
           void answer()
                {message_for_caller.on();
                 message_for_caller.put_data(information);
                 message_for_caller.off();
                 recorder.on();
                }
           void play()
                {message_from_caller.on();
                 message_from_caller.put_data(my_list.head());
                 display_position();
                 message_from_caller.off();
                 my_list.dequeue()
                }
       } /* end answering machine */
main(){ };
```

The concept of *devices as class hierarchies* provides a simple parallel performance of the methods. Note, synchronization is not supported in this hierarchy and must be provided through other components; multiple inheritance is often needed.

16.6.4 Processing Units as Classes

This abstraction comprises source objects, destination objects and combined source-destination objects which perform intermediate processing of continuous data. With

this approach, a kind of "lego" system is created which allows for the creation of a data flow path through a connection of objects. The outputs of objects are connected with inputs of other objects, either directly or through channel objects.

As an example of this concept, the *processing unit* as a class is presented. (It was originally implemented as part of the DiME project of the IBM European Network Center in Heidelberg, Germany [SHRS90].). It was used at the beginning of the application implementation of a remote camera control system in Heidelberg. It should be understood that it is as an example only.

Similar considerations are discussed in [AC91] with the node types of COMET, in [GBD+91] with sources, destinations and filters, and in [SS91] with modules of a variable number of input and output channels.

Multimedia Object

A multimedia application processes and controls (respectively generates) the interactions and information of different continuous and discrete media. From the object-oriented viewpoint, an application is considered to be a *multimedia object.* Such an object uses or consists of many other objects which contribute to the solution of the task. These objects are connected to, for example, representation of different media and device units. Such a *Compound Multimedia Object (CMO)* consists of other CMOs and *Basic Multimedia Objects (BMOs).* The *Basic Multimedia Class (BMC)* typically represents an individual medium of an input type (e.g., data from a camera, stored audio sequences from a file) or output type (e.g., data output to a video window or speaker). A *Compound Multimedia Class (CMC)* can control, and respectively represent several media and devices. BMOs are instances of BMCs; CMOs are instances of CMCs.

Data can be either *transient* or *persistent.* For example, if an image is read from a hard disk, the following properties are connected with each other: *life span, input type* and *medium* image. Generally, the media are *text, image, audio* and *video.* Further, it is possible to consider other media such as *tables* or *drawings* which need to be included in the range of BMCs.

For clarification of the properties of this approach, an encyclopedia example is discussed below. Instead of presenting the example in an object-oriented language, the example specification is chosen in an easy-to-understand notation for didactical reasons:

```
Lexicon: compound_object;
     DATA: Explain external;
           Animation external;
     ACCESS_POINTS: VIDEO_SOURCE Animation.VIDEO_SOURCE;
                    AUDIO_SOURCE Animation.AUDIO_SOURCE;
                    TEXT_SOURCE Explain.TEXT_SOURCE;
     METHODS:
           start: display(Explain);
           play: play(Animation);
           pause: pause(Animation);
           stop: stop(Animation);
           ...
Animation: compound_object;
     DATA: Speech external;
           Scene external;
     ACCESS_POINTS: VIDEO_SOURCE Scene.VIDEO_SOURCE;
                    AUDIO_SOURCE Speech.AUDIO_SOURCE;
     METHODS:
           play: play(Speech), play(Scene) in_parallel;
           pause: pause(Speech), pause(Scene) in_parallel;
           stop: stop(Speech), stop(Scene) in_parallel;
           ...   ...
     EVENTS:
           audio_end: wait(video_end); stop(Scene);
           video_end: wait(audio_end); stop(Speech);
 Scene: basic_object;
     DATA: VIDEO_filename at node_1;
     ACCESS_POINTS: VIDEO_SOURCE;
     METHODS:
```

```
                play;
                pause;
                top;
                ...
          EVENTS:
                at_end: display(LAST_PICTURE), inform_PARENT(video_end);
                ...   ...
      Speech: basic_object;
          DATA: AUDIO_filename at node_2;
          ACCESS_POINTS: AUDIO_SOURCE;
          METHODS:
                play:
                pause:
                stop:
          EVENTS:
                at_end: inform_PARENT(audio_end);
      Explain: basic object;
          DATA: TEXT_filename at node_1;
          ACCESS_POINTS: TEXT_SOURCE;
          METHODS:
                display;
                ...
          EVENTS:
                ...
```

In the above example, the actual classes of the lexicon are presented. Instead of *Explain: basic_object*, actually *Explain: basic_class* should be given; instead of *Animation: compound_object*, *Animation: compound_class* should be considered. The reason why objects instead of classes were considered is just simplicity. This way, the example shows its run-time behavior better. BMOs are used as processing elements with the sources, destinations and combined sources-destinations of media streams. Corresponding to the *class hierarchy of BMOs*, different properties can be chosen at the highest level of the classification.

In Figure 16.3, a division of source, destination and combined source-destination is

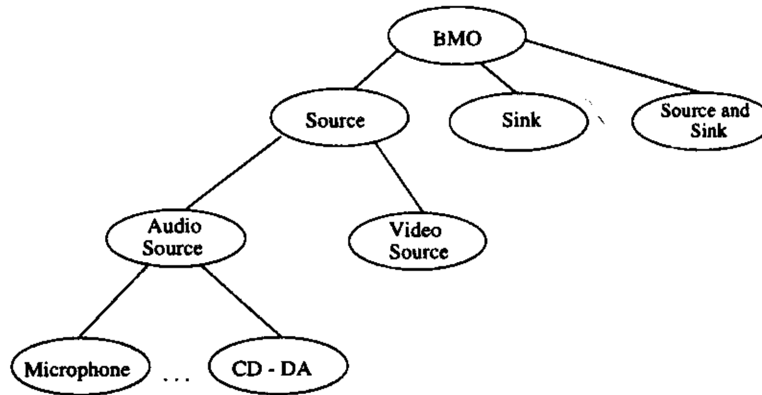

Figure 16.3: *Example of a multimedia class hierarchy.*

shown as the primary qualification attribute. The medium is defined as a secondary attribute. It is important to point out that in other examples, several BMOs from the same BMC can coexist simultaneously.

Data Specification DATA

The *data specification* DATA specifies the BMO's binding to a file, a device unit and their place. A *device unit* mostly consists of hardware and software, although a device unit may only consist of a software module. There can be objects at different places which point to remote BMOs. In the data specification, the lexicon example shows the binding of the object *Scene* to the *VIDEO_filename* at the place *node_1.*

Methods

Each BMO contains a set of *methods* that is dependent on its class. Some methods, which repeat themselves, are *play* and *stop.* The internal state of all multimedia objects which process continuous media is constantly renewed with current values because the actual data (for example, a picture of a video scene) are valid only during a certain time interval. Therefore, some methods also support time-dependent functions such as *slow motion*, *presence* of a scene during two seconds or *volume*

increase during five seconds.

Event Processing

BMOs include event-driven processing to inform other objects about certain states of their own objects, for example, *start* and *end*. Further, they are used to transmit asynchronous events which happen during the life span of an object. Therefore, events are special kinds of operations of an object. They can be compared to *exceptions* in real-time programming languages such as CHILL. A BMO, connected to a continuous data stream, contains in our model its own defined time-scale and events. A priori-defined events are *start* and *end*. Additionally, there can be defined user-specific events. The relation between events and their event processing is specified for each BMO.

Access Points

Continuous data of a data stream can be processed from several objects consecutively. The corresponding BMOs must be connected with each other. Therefore, each BMO contains one or several *ports* which are called *access points*. Access points for BMOs are objects which consist not only of local addresses, but also of protocols with entries concerning the required data coding and compression. They include all source- and destination-specific information which is necessary for their connection. BMOs can have access points such as *VIDEO_SOURCE*, and *VIDEO_SINK*. The application can pass the multimedia data to storage, communication, presentation and processing units through access points.

Transparent to the manipulation of multimedia objects, is the control of adequate transmission over the network(s). The network component takes the necessary information, including the QoS parameters for connection setup and for data transmission from the object-control-information. The required resource management is interfaced by this processing.

Channel Object

A binding between one or several sources and one or several destinations can be implemented. A *channel object* is created with dedicated and communication-supportive methods to bind sources and destinations. After creations of such a channel, sources and destinations can be bound to each other through a *bind* call to the particular channel. This approach allows the implementation of not only multicast, but also n-to-m connections ($n, m > 0$). The access rights of the particular object are assigned during the *bind* call. Destinations and sources can be bound to the same channel independently through this mechanism.

Depending on the channel object implementation, one or several sources can be connected to it. Usage with an individual source is the same as in the case of a normal TV transmission channel: a channel is created, the data of the source object are recorded and each authorized user can connect to this channel and can receive the continuous data.

In a general case with several sources, the channel serves as a mixer of data and it must also guarantee synchronization. Using channel objects, it is possible to dynamically change the bindings among different multimedia objects. They always contain the methods *connect* and *disconnect*. To fully integrate BMOs into the system, this multimedia connection management must be implemented as a multimedia communication system.

CMO, BMC and CMC

Given that different media are not only supported separately, but also simultaneously, some device-dependent media combinations must be integrated into the object-oriented model.

An application should additionally find predefined *Compound Multimedia Classes (CMCs)* and their corresponding CMOs, as well as create them itself. In this section, this extension of the model is discussed.

An application implements a new CMC by using content-containing BMCs and

CMCs. This new class contains other classes inside its data specification, access points, methods and/or event processing. With respect to methods, this property is called inheritance.

The *activation* of a CMC creates a CMO. A binding inside of the data specification can be set inside the class, as well as for a specific object. Additionally, information, such as life span, access rights and distribution aspects can be assigned to an object.

The *methods of CMOs* can be set up application-specifically. They are developed using methods of content-containing BMCs and CMCs and private algorithms. In our example, the method *play* of the CMO *Animation* uses the methods *play* of the objects *Speech* and *Scene*.

Similar to the method specification, the *event processing of CMOs* is defined. Typically, either methods of the *content-containing* object (see *stop(Scene)* of the CMO *Animation*) are called or an event is sent to other objects (see *inform_PARENT (video_end)* of the CMO *Scene*). Events inform the called objects about the state of the calling object.

The use of methods and events allows the application to create a script which expresses the interactions of different objects precisely and relatively simply. Thus, the presentation of multimedia data can be determined.

Distribution of BMOs and CMOs

Another aspect is the *distribution of BMOs and CMOs*. A CMO can consist of objects which are distributed over different computer nodes.

The channel object with the methods *create_connection* and *delete_connection* supports the possibility to manage connections of several media together. Internally, the transmission can be implemented either through integration and interleaving of different media over one connection or through several individual connections. Thus, the access rights of data can be examined. For example, consider the class of a CMO, called *lexicon*, as shown in Figure 16.4. BMOs and CMOs (the same as BMCs and CMCs) contain the four areas *DATA, ACCESS_POINTS, METHODS* and *EVENTS*. This example shows BMOs, as well as CMOs. The data of the BMO

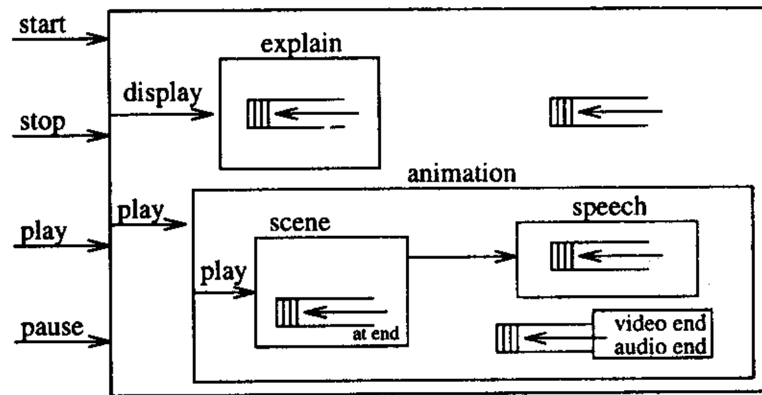

Figure 16.4: *Example of a CMC.*

specification are abbreviated with *_filename*. At the moment when the CMO *lexicon* is activated, or started, a text of the text object *Explain* is displayed. The user can call the method *play* in the application which activates the corresponding method in the CMO *Lexicon*. *Play* starts the presentation of a combined audio-video sequence of the object *Animation*. The application also allows the user to access methods like *pause* or *stop*. Thus, the corresponding method in the lexicon is called. The CMO *Animation* is a composition of audio and video objects.

If one assumes that the audio and video information output do not posses the same length, then certain actions must be performed when one of the media finishes earlier than the other(s). The method *play* of the object *Animation* starts the video scene and the audio passage concurrently. If the audio passage finishes before the video scene, the event *audio_end* is sent from the object *Speech* to the object *Animation*. If the video scene finishes before the audio passage, the last image of the video sequence is displayed and the signal *video_end* is sent to the object *Animation*.

All events of the object *Animation* cause a data flow to be stopped, some media-dependent methods to be performed, and the waiting of an event of another object. At the end, the combined audio-video information as a whole unit is stopped. This kind of synchronization is known as *conditional blocking* [Ste90].

In the preceding paragraphs, *sources* were specified as BMOs and CMOs. In the same way, destinations and combined source-destination objects can also be specified

as BMOs and CMOs. They may represent:

- *Output devices* such as windows, monitors or speakers.
- *Files* of internal secondary storage devices or external storage devices.
- *Processing units* of continuous media with input and output ports (hardware and/or software).

The above described model served as the basis for the development of a remote camera control [WSS94]. This application (remote camera control), implemented in C++, includes communication support which allows the control of different kinds of cameras from a remote place. In another recent approach at the EPFL (Lavosanne), a language, called *Sync C++* has been implemented as a concurrent version of C++ [CDP94]. There, Petitpierre defined active objects which were used for real-time programming of multimedia data. These objects had their own life span (as threads of execution) and hence, worked independently from the others. Sync C++ was shown to be very efficient in writing multimedia applications with a large amount of user interface handling.

16.6.5 Media as Classes

The *Media Class Hierarchy* defines a hierarchical relation for different media. The following example shows such a class hierarchy. Thus, the individual methods of the class hierarchy will not be described. The class *Pixel* in the class hierarchy uses, for example, multiple inheritance.

```
Medium
     Acoustic_Medium
       Music
         Opus
           Note
             Audio_Block
               Sample_Value
```

```
      Speech
         ...
       ...
    Optical_Medium
      Video
        Video_Scene
          Image
            Image_Segment
              Pixel
            Line
              Pixel
            Column
              Pixel
     Animation
       ...
     Text
       ...
     ...
   ...
   other_continuous_medium
   discrete_medium
```

Another example shows the following class hierarchy (note, so far we have not found a *best* class hierarchy because different class hierarchies are better suited for different applications):

```
Medium
  continuous_medium
    Audio
      Audio_Passage
        Music
        Speech
        Noise
   Video
```

```
    Video_Scene
  Animation
    Animation_Scene
  Pointer_Information
  Further_Realtime_Data
discrete_medium
  Image
    Image_Segment
      Pixel
    Line
      Pixel
    Column
      Pixel
  Text
  Formula
  Table
    Line
      Attribute
      Formula
      Value
  ...
```

A specific property of all multimedia objects is the continuous change of their internal states during their life spans [SHRS90, GBD+91]. Data transfer of continuous media is performed as long as the corresponding connection is activated. A *connection* can be either a connection for local data transfer between source(s) and destination(s) (e.g., from a disc to a video window), or a connection for remote data transfer. Data can also be transmitted when a method does not exist for the object (besides the methods *new* and/or *init*), respectively a message was not sent. The activation happens implicitly in this case. Thus, the internally managed storage areas of continuous media always take new values. This is also valid for the control information, for example, time stamps.

Besides the class hierarchy, the main attributes (expressed or retrieved through methods) need to be considered for different classes. Typical methods used by all

continuous media are, for example, *play* and *stop*. We have implemented these methods for different classes and different devices. The lesson learned is that it is relatively easy to enforce the same or similar semantics for different implementations.

Some programming environments already provide languages based on the object-oriented approach with processing units or media as classes. For example:

- *Gibbs' Multimedia Programming Environment* has a strong media relation [GBD+91]. It is implemented through a *scripting language* extension at the user interface. This language belongs to the category *processing units* as classes. Here, constructs for input of parallel, sequential, repetitious processing are included ($a >> b$, a & b, $n \times b$).
- The *Rendezvous*™*Environment* includes graphical classes [HBP+93]. The primitive graphical classes include the full range of drawing primitives available to the X Window System™, such as lines, rectangles, polygons, arcs, ovals, text and color and monochrome images. In this environment, the application programmer uses the *Rendezvous Language*, which is an object-oriented language extended with features that simplify the construction of multi-user interfaces using the Rendezvous Architecture. The structure of the major Rendezvous Language components, including the interface to the X Window System via CLX (Common Lisp X), is shown in Figure 16.5.

Considering media as classes is a general approach. As a part of each application there can also be further multimedia-specific class hierarchies.

16.6.6 Communication-specific Metaphors as Classes

Communication-oriented approaches often consider objects in a *distributed environment* through an explicit specification of classes and objects tied to a communication system. Blakowski specifies, for example, information, presentation and transport classes [Bla91a]. The information, contained in the information objects, can build a presentation object which is later used for presentation of information. Information objects can be converted to transport objects for transmission purposes (see [Bla91a] for the complete state-transition graph). Possible extension to this model would be

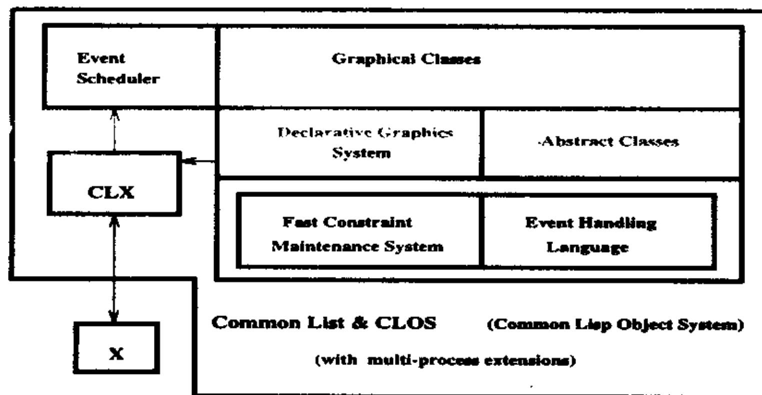

Figure 16.5: *Structure of Rendezvous Language components [HBP+93].*

a *storage class.* Information is often processed differently. It depends on whether the information should be presented, transmitted or stored. With storage objects, it is necessary to consider the different storage formats. Relevant formats are the coding and compression formats, format of interleaved data streams and formats such as CD-ROM ISO 9660.

16.7 Comments

The current research in programming of multimedia applications leads to the following conclusions:

- There are too few proper data abstractions for structural programming of multimedia data.
- The abstraction level for *higher procedural programming languages,* which is used for the programming of continuous media has been considered only to a very small extent.
- There is a great number of object-oriented and toolkit approaches based on very different class hierarchies. Consolidation is needed.

Hence, the following comparisons can be made:

- *Libraries* represent the simplest integration of multimedia devices and functions in a system. The functions of such a library can be called from the system software level, as well as by a programming language. Different devices (often with the same functionality) provide different interfaces. Hence, portability of multimedia programs tends to be difficult.

  Using this library approach, multimedia products can be brought to market in a short time with *new and already improved functions.* However, after some time, the complexity of the application development increases because of different hardware which slows down the further spread of this approach. Therefore, other abstraction levels are necessary.

- *System software interfaces* integrate multimedia functions into the operating system. Currently, approaches are tied to specific operating systems and to one kind of multimedia data processing.

  Instead of providing devices in the form of libraries for processing of continuous media, it is necessary to integrate device drivers into the operating system. The main problem is current operating systems: most of them do not provide real-time processing. Today, it is still necessary to find tricky ways to provide multimedia processing.

- *Higher procedural programming languages* can rely on library functions when continuous media programming is integrated. However, these libraries are product-specific and very different. Therefore, a corresponding programming language needs to communicate with several libraries. This requires a high development effort.

  As an alternative, it is possible to access abstractions such as *constraints* and *event handlers* in the operating system. However, because this alternative is available only in special cases (e.g., the Rendezvous Environment and Nemo system), this approach will take time to catch on. Thus, it is not clear how these abstractions will differ in various operating systems. The development of current commercial multimedia applications is done mostly using higher procedural programming languages. Here, the functions of existing libraries can be called directly. Integrated programming of applications with continuous media inside higher programming languages leads to simpler and clearer programming.

- *Object-oriented approaches* are the most widespread abstractions for research in programming of multimedia applications today. Existing system software is most often used for integration with physical devices. Therefore, as a first step, all functions of this system software is tied to its class hierarchy. The same happens with communication systems. Thus, the developed class hierarchies are very different, and one cannot currently identify the *best* class hierarchy. The combination of the media-related class hierarchy with the "lego" approach of processing units may satisfy several applications.

There is no *best* approach to the abstraction of multimedia data (according to current experiences and results). Different abstraction levels will coexist; however, to mutually use the respective results, these layers must build on one another (as shown in Figure 16.1).