Proceedings, 3rd IEEE International Conference on Image Processing (ICIP-96), Lausanne, Switzerland, September 1996.

A SYNTACTIC FRAMEWORK FOR BITSTREAM-LEVEL REPRESENTATION OF AUDIO-VISUAL OBJECTS

Yihan Fang and Alexandros Eleftheriadis

Department of Electrical Engineering
and Image Technology for New Media Center
Columbia University, New York, NY 10027, USA
Email:{fang, eleft}@ee.columbia.edu

ABSTRACT

We present the design of a novel syntactic and semantic framework to describe the bitstream-level representation of audio-visual information objects. This framework is important for emerging techniques in video compression, and is currently part of the MPEG-4 System Description Language Working Draft. The design is based on a principle of orthogonality between parsing and processing, i.e., separating the bitstream parsing from the other decoding operations, and extends the basic typing system of a programming language so that it incorporates sophisticated parsing information. Examples of object representations are provided using the MPEG-2 Video bitstream syntax.

1. INTRODUCTION

In the past several decades, the fundamental principle of digital audio-visual information representation has been compression. The diversity of the potential applications that use video and audio, however, makes it extremely difficult to prescribe a single, universally applicable compression approach. Emerging applications such as videoconferencing over computer networks and general computer-supported collaborative work environments, indexing and selective retrieval, classification and archiving, editing and post-processing have clearly shown that there are several challenges that cannot be tackled with today's standards' designs. As a result, special purpose techniques have been developed to tackle some of these problems, be operating directly in the compressed signal domain (e.g. DCT-based filtering, compositing, dynamic rate shaping, etc.). Their complexity, however, makes them less than an ideal approach. Use of a flexible "programming" framework capable of describing video processing algorithms and being executed on different platforms will facilitate the development of these and other applications.

This is one of the main themes in the on-going ISO/IEC MPEG-4 standardization effort. It is intended to offer more flexibility and extensibility than traditional approaches. MPEG-4 is addressing, among other things, the development of a System Description Language (MSDL), that can describe both the bitstream syntax as well as the decoding tools or algorithms. This way, audio-visual information objects can be encoded using the best technique suitable for the particular application at hand. If the technique is not known to the decoder, it can be downloaded prior to the actual data. This of course requires that the decoder is able to execute code in that particular language, but it allows complete customization of the decoding process. We are addressing a more general syntactic and semantic framework. In addition to the description of decompression algorithms, our approach is also aimed on the description of audio-visual compression, as well as processing and manipulation. In this paper, we address the issues of bitstream-level representation, which are common in all cases. Our approach originates from the C-like syntax that has been successfully used to describe the structure of coded audio-visual components in MPEG-1 and MPEG-2, with the addition of object-orientation and thoroughly defined semantics suitable for machine translation. This work is currently part of the MPEG-4 MSDL Working Draft [6].

2. THE FORMAL LANGUAGE FOR AUDIO-VISUAL OBJECT REPRESENTATION

Our approach is called the Formal Language for Audio-Visual Object Representation (FLAVOR). FLAVOR is focused on accommodating novel and conventional functionalities for audio-visual coding. It should be

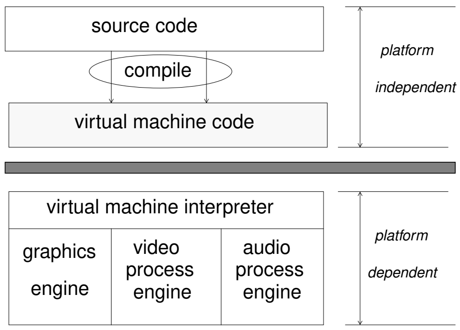

Figure 1: Virtual Machine

able to handle both syntactic (bitstream-level representation) and algorithmic descriptions (procedures operating on data). For algorithmic descriptions, the process is rather involved. The need for platform independence can be satisfied by using a virtual machine (VM) architecture. Like other programming languages using virtual machines, most notably Java [7, 8], the language would have the advantage of being architecture-neutral. By having virtual machine interpreters implemented on different platforms, users would not need to compile their programs for each individual platform; the virtual machine interpreter will directly execute the binary VM source code. Figure 1 shows the FLAVOR virtual machine structure. For developing a new generation, the design of a syntactic representation framework is the first step. In the next section, we provide the bitstream syntax representation of FLAVOR.

3. DATA REPRESENTATION

The bitstream-level representation of FLAVOR is not defined in a manner of particular procedural implementation (i.e., a parser), but rather as a "syntax" specification. In other words, one should not describe the parsing process (an implementation issue) but only the way data are mapped on the bitstream. This enables the use of different implementations to parse and process individual objects, optimized for the particular architecture. An additional benefit is that it is generic and amenable to conversion to a simple binary representation.

The syntactic representation framework we propose has two different levels: a textual one, that associates coded information with meaningful names, as well as a binary one that denotes the actual bits that are placed in the bitstream. These bits need to be sent to the "decoder", so that it knows how to parse objects of the specific type later when they are sent to it. Here we focus on the textual representation only.

3.1. Elementary Data Types

We can in general identify the following syntactic elements:
1. *Constant Length Direct Representation Bit Fields.*

```
type[(length)] element_name[=value];
```

For example, an entity such as temporal reference would be represented as:

```
unsigned int(5) temporal_reference;
```

where `unsigned int(5)` indicates that the element should be interpreted as a 5-bit unsigned integer. Two non-standard data types are `bit` (just a sequence of bits) and `vlc`, which we discuss below.
2. *Variable Length Direct Representation Bit Fields.* This is the case where the length field is determined by a variable. For example:

```
unsigned int(3) precision;
int(precision) DC;
```

3. *Constant Length Indirect Representation Bit Fields.* The length of the bit field can be mapped to the actual values using mapping algorithm. This is accomplished by defining `map`:

```
map map_name (type_idx, type1,..., typeN){
    index value1 ... value_M,
    [index value1 ... value_M, ...]
};
```

4. *Variable Length Indirect Representation Bit Fields.* In this case, we have the data type `vlc` as the index data type. For example:

```
map vlc_table (vlc, int value){
    0b0000.001     0,
    0b0000.0001    9
};
```

The interpreter would look up for `vlc_table` in map definition when it encounters the bit field of `vlc_table`. Then it will map the variable codewords to actual values automatically.

3.2. Composite Data Types–Objects

Composite types or objects are defined in classes. This approach follows an object-oriented paradigm and thus allows for mapping both traditional as well as more advanced "object-based" representations. Object orientation is essential in FLAVOR, since it facilitates the definition of interfaces between self-contained entities (audio-visual objects), and allows for reusability.

```
[modifiers] class name [(parameters)] [is parent]
     [:[aligned] bit(length) [id_name]=id|range]{
   [element; ... ]    // zero or elements
};
```

An *element* can be either a fundamental-type data element as described in Section 3.1, or a composite-type object, which is an instance of another class. The order of declaration of bitstream components determines the order in which the elements appear in the bitstream. Similar to most modern object-oriented programming languages, our language would support a hierarchical structure. The `id` (or `range`) above, when specified, is placed at the beginning of the object in the bitstream, to facilitate demultiplexing of different objects to their corresponding processing functions.

We introduce "parameterized class" to represent a class with parameter(s). This is to address cases where the data structure of the class is dependent on some variables of other objects. A parameterized class is dependent on the objects (could be class objects or member variables) in its parameter list. A parameterized class cannot be instantiated without first instantiating the required dependent classes. It will be instantiated automatically when a "compositing" class (a class that contains an instance of this class) is instantiated.

3.3. Syntactic Control Flow

The syntactic control flow provides constructs that allow conditional parsing, depending on context, as well as repetitive parsing. Objects that are sensitive to context or defined repetitively are very common in existing audio-visual processing algorithms. For instance, in the MPEG-2 standard, *macroblock* objects are defined differently for different chrominance formats. The conventional implementation of context is by procedures. In our framework, we consider extremely essential to avoid the use of procedural declarations for defining objects, and instead use conditional constructs to accommodate context. The familiar C/C++ `if-else` construct is used for testing conditions, and the `for`, `do` and `while` constructs are used for repetitive definitions. The following example uses `if-else` control flow to construct a conditional object:

```
class conditional_object {
    unsigned int(3) foo;
    int(1) bar_flag;
    if (bar_flag) {
         unsigned int(8) bar;
    } else {
         map(some_vlc_table) bar;
    }
    unsigned int(32) more_foo;
};
```

The data structure of this `conditional_object` appearing in the bitstream is depicted in Figure 2. In this

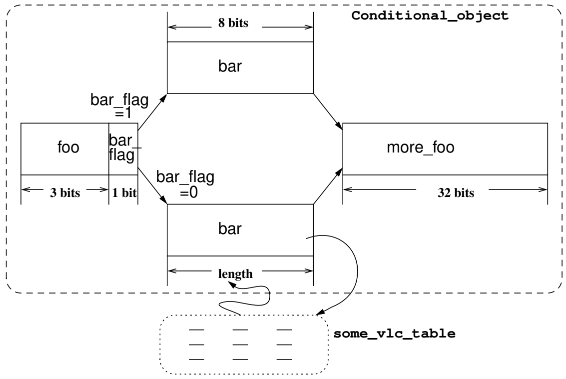

Figure 2: Bitstream data structure for conditional_object class example

example we allow two different representations for `bar`, depending on the value of `bar_flag`. Note that the use of a flag necessitates its declaration before the conditional is encountered. The construct `[bitstring]` is a test condition that is true (non-zero) if the next bits present in the input bitstream are equal to bitstring. The construct `[bitstring*]` performs the same operation, but if the string is found, the bits are removed from the bitstream.

3.4. Temporary Variables

In order to accommodate complex syntactic constructs, in which context information cannot be directly obtained from the bitstream but is the result of a non-trivial computation, a small number of temporary variables are provided. These are named `$a` through `$z`, and can hold any variable type up to a maximum length of 128 bits. They can be used in expressions and conditions in the same way as bitstream-level variables. The following example illustrates the need of using temporary variables when the number of non-zero elements of an array is computed.

```
unsigned int(6) size;
int(4) array[size];
for ($i=0, $n=0; $i<size; $i++) {
    if (array[$i]!=0)  $n++;
}
int(3) coefficients[$n];
// read as many coefficients as there are
// zero elements in array[]
```

4. OBJECT PARSING

Parsing objects from the audio-visual bitstream is a very important practical implementation issue. An object is considered parsable from the bitstream if at least one member of the class is parsable.

For each parsable class, there is a built-in public data member `parsed` that indicates whether an instance of this class has been parsed from the bitstream. When an instance of a parsable class is parsed from the bitstream, the data member `parsed` would be set to 1. This is useful in the need of checking whether an object is present in the bitstream. Other potentially useful built-in variables can be defined as well.

A parsable object is guaranteed to be parsed from the bitstream before being accessed. This gives the maximum flexibility in terms of implementation, allowing parsers to follow the best parsing strategy for their particular architecture.

5. REPRESENTATION OF MPEG-2 VIDEO SPECIFICATION

Using the above simple framework, we have represented the entire MPEG-2 video bitstream syntax specification [4]. An example of the `slice` class in MPEG-2 is as follows:

```
class slice(sequence_header sh,
            sequence_extension se,
            sequence_scalable_extension sse,
            picture_spatial_scalable_extension psse,
            picture_header ph,
            picture_coding_extension pce):
        const bit (32) slice_start_code =
                  0x00000101..0x000001AF {
    if ((sh.vertical_size_value +
         se.vertical_size_extension<< 12) > 2800)
        unsigned int(3) slice_ver_position_ext;
    if (se.parsed) {
        if (sse.scalable_mode == 0b00)
            unsigned int(7) priority_breakpoint;
    }
    unsigned int(5) quantiser_scale_code;
    if ([1]) {
        bit(1) intra_slice_flag;
        unsigned int(1) intra_slice;
        unsigned int(7) reserved_bits;
        while ([1]){
            unsigned int(1) extra_bit_slice=1;
            unsigned int(8) extra_info_slice;
        }
    }
    unsigned int(1) extra_bit_slice=0;
    do{
        Macroblock(se, sse, psse, ph, pce) mb;
    }while(![0b0000.0000.0000.0000.0000.0000.0001])
};
```

6. CONCLUSION

The Formal Language for Audio-Visual Object Representation (FLAVOR) is designed to provide a flexible environment for the development for audio-visual information, coding and compression/decompression algorithms. We have presented the design of the bitstream-level data representation framework. Its key features are the separation of syntax specification from the actual parsing/processing operation, and its independence from the actual language to be used. The approach follows an object-oriented paradigm, thus being capable of mapping both traditional as well as more advanced "object-based" representations. A translator to C++/Java is already being implemented. Future work includes the design and implementation of binary version of syntax, the design of procedural language, as well as the design of virtual machine.

Acknowledgments

The authors would like to thank the entire MPEG-4 MSDL Subgroup for useful discussions, and especially Drs. P. Chou and J.C. Dufourd.

7. REFERENCES

[1] Y. Fang and A. Elefttheriadis, "The MPEG-4 Syntactic Description Language", submitted to Image Communications.

[2] A. Eleftheriadis and Y. Fang, "A Revision and Proposed Grammar for MSDL-S", ISO/IEC JTC1/SC29/WG11 Contribution M0856, Florence, Italy, March 1996.

[3] A. Eleftheriadis, "A Syntactic Description Language for MPEG-4". ISO/IEC JTC1/SC29/WG11 Contribution M0546, Dallas, TX, November 1995.

[4] "MPEG-2 Video", Recommendation ITU-T H.262 (1995 E), ISO/IEC 13818-2 International Standard, 1995.

[5] AHG on MSDL issues, "Requirements for the MPEG-4 SDL(Draft in Progress Revision 1.1)". ISO/IEC/JTC1/SC29/WG11, Tokyo, July 1995.

[6] *MSDL Specification. Version 1.1*, ISO/IEC JTC1/SC29/WG11 N1246, Florence, Italy, March 1996.

[7] "The Java Language: A White Paper", Sun Microsystems, 1994.

[8] "The Java Language Specification", Release 1.0 Alpha3, Sun Microsystems Computer Corporation, May 1995.