INTERNATIONAL ORGANIZATION FOR STANDARDIZATION
ORGANISATION INTERNATIONALE DE NORMALISATION
ISO/IEC JTC1/SC29/WG11
CODING OF MOVING PICTURES AND ASSOCIATED AUDIO

ISO/IEC JTC1/SC29/WG11
MPEG95/M0546
November 8, 1995

Source : Alexandros Eleftheriadis, Columbia University
Title : A Syntactic Description Language for MPEG-4
Purpose : Proposal
Subgroup : AOE/MSDL

1. Introduction

This document proposes a concrete starting point for a syntactic representation framework for audio-visual objects, as embodied in MSDL. It originates on the C-like syntax that has been successfully used to describe the structure of coded audio-visual components in MPEG-1 and MPEG-2. This syntax has two different levels: a textual one, that associates coded information with meaningful names (e.g., picture_start_code), as well as a binary one that denotes the actual bits that are placed in the bitstream. MSDL's first and foremost purpose is to describe coded audiovisual information; as such, it must associate bit-level representations with meaningful quantities, and describe the ways these can be combined together to form valid bitstreams. For this purpose, using the existing framework of MPEG-1/2 is deemed as a very suitable starting point.

The features added to the old scheme include object-orientation and more thoroughly defined semantics that can be used for potential machine translation. Even though the current short-term focus in MSDL is Level-1 (L1) functionality (numbering levels from 0 to 2), the long-term goal is to provide full Level-2 (L2) capabilities; care must then be taken in the syntactic description to allow this to happen. The primary difference between L1 and L2 is that in the latter one can specify new tools. This necessitates an MSDL that can actually describe new syntactic components (e.g., an octagonal block) and their processing (or decoding) algorithms, hence making it an application-specific programming language.
Hence MSDL must be able to describe the following three items:

1. syntax (similar to MPEG-1 and MPEG-2; sufficient for L0 and L1),
2. syntactic description (meta-data, used to convey the structure of new syntactic elements to the decoder, for example the octagonal block; needed for L1 and L2),
3. algorithm description (the algorithmic part of the decoder; needed only for L2).

This document primarily describes the data representation aspects of this approach, i.e., how can one describe the data that are placed in the bitstream. If, however, one wants to add L2 functionality, it is straightforward to continue and add items 2 and 3. The exact process to do is succinctly outlined. For item 2, it is simply a matter of using a bit-efficient binary representation of the formalism used for item 1. For item 3, the process is more involved, and necessitates the use of a virtual machine.

It is important to note that this framework is in no way restricted to A/V objects in their classical sense.
In fact, it can describe any structure that is represented by a series of bits. This includes synthetic objects based on 2-D or 3-D graphics models. Concreteness is necessary in order for MSDL to become useful in specifying coding tools and algorithms, while generality is needed so that unanticipated developments can be easily accommodated. It is worth noting that the framework described here can describe the entire current set of MPEG specifications.

We should point out that this document describes work in progress, and should be considered as such.

2. Syntactic Features

In general, all coding approaches recognize specific units on which they operate. Taking MPEG-2 video as an example, we have the segmentation of sequences to frames/fields, slices, macroblocks, blocks, etc. Each such unit can be considered as an object (albeit dependent on other objects). The word object is used here in the loose sense of a semantically important unit. Some objects are placed in the bitstream after an appropriate object ID (e.g., slices); others, for reasons of efficiency, do not have such ID (e.g., MBs and blocks). Within each of these objects we encapsulate some ancillary attributes. For example, slices contain their vertical position within their ID (start code), MBs contain address increments, etc. This suggests a quite natural approach in constructing a data hierarchy, that has been proposed in the past in other MPEG contributions that casted the MPEG-2 syntax in an object framework [1] and mapped the MPEG picture structure into C++ [2]. Here we attempt to make such a representation more economical and concrete, and also address the needs of representing arbitrary binary information.

We can in general identify the following syntactic elements:
1. Constant-length direct representation bit fields (e.g., temporal_reference[1]). These include the encoded value as it is to be used by the decoder.
2. Constant-length indirect representation bit fields (e.g. chroma_format or coded_block_pattern). These require an extra lookup into an appropriate table, or some algorithmic processing to obtain the desired value (e.g., coded_block_pattern).
3. Variable-length bit fields (Huffman codes). These, by definition, always correspond to indirect representations.

Note that the distinction between 1 and 2 is important only for a L2 MSDL.

In addition to these elements, we have to deal with context: some bitstream elements are present only for specific values of others. These elements act as “flags,” and are heavily used in practice.

In the following, we present an approach to specify the above elements, starting from individual ones and then showing how to build larger components.

---

[1] MPEG-2 video entities are used in the cited examples.

3. Fundamental Types

3.1 Constant-Length Direct Representation Bit Fields

Following the approach used in MPEG-1/2, these can be simply represented as:

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

The *value* attribute is only present when the value is fixed (e.g., start codes or object IDs). The *type* and the optional *length* are always present, except perhaps when length is 8 or 16 (some default value). As an example, a start code would be represented as:

const(32) some_start_code=0x000001B1;

An entity such as temporal reference would be represented as:

uint(5) temporal_reference;

where uint(5) indicates that the element should be interpreted as a 5-bit unsigned integer (by default with the most significant bit first—MSBF).

3.2 Constant-Length Indirect Representation Bit Fields

Specification of such fields in detail is important only for a L2 MSDL. For L1, it is sufficient to specify
the element as an integer: it is up to the tool's processing algorithm to interpret the element as needed, and for L1 such an algorithm is not specified using MSDL.

Assuming that we are interested at L2, in addition to the actual element we need to define how it is mapped to obtain the actual values that the decoder will use. This can be accomplished by defining the map itself:

```
map(type_idx, type1, type2, …, typeN) MapName {
        index value_1 … value_M,
        [index value_1 … value_M, …]
};
```

Note that this is a multi-dimensional mapping: there is one input (index), and several outputs (values), allowing the table to be used to specify the value for several different variables (VLCs typically encode several events jointly for efficiency reasons). The *type* entries in the definition indicate the type of the value columns. The *index* entry is always specified as a series of bits.

As an example, we have:

```
// a table that relates the chroma format with the number of blocks per signal component
map(uint, uint, uint, uint) blocks_per_component {
        0       4       1       1,      // 4:2:1
        1       4       2       2,      // 4:2:2
        2       4       4       4       // 4:4:4
};
```

Multiple input maps are primarily useful when joint encoding is performed; we are not aware of any use of them in decoding schemes. If necessary, the definition can be trivially extended to cover this case as well.

3.3 Variable Length Bit Fields

For a variable length element utilizing a Huffman table, the following specification is used:

vlc(*table*) ac_dct_coefficient;

The definition of the table is done in exactly the same way as we saw before. For example:

```
map(vlc, boolean, int) sample_vlc_map {
        0000.001        0       5,
        0000.0001       1       -14
};
```

The vlc codewords are binary strings, optionally using the period ('.') every four digits for readability. Boolean variables are simply 1 bit entities (equivalent to int(1)).

Very often, VLC tables are incomplete: due to the large number of possible entries, it is inefficient to keep using variable length codewords for all possible values. This necessitates the use of escape codes, that signal the subsequent use of a fixed-length representation. To allow for such exceptions, element specifications are allowed for map values. This is illustrated in the following example:

```
map(vlc, uint) sample_map_with_esc {
        0000.001        0               5,
        0000.0001       1               -14,
        0000.0000.1     boolean         int(32),
        0000.0000.0     0               -20
};
```

As written above, when the codeword 0000.0000.1 is detected in the bitstream, then the next bit will correspond to the value for column 1, while the following 32 bits will correspond to the value for column 2 (the order is significicant). Using this construct, the complete behavior of the VLC mapping is described in a concise manner in a single place. Note also that cascaded VLC tables are allowed: simply substitute, for example, 'boolean' by 'vlc(another_table)'.

A final trick that is present in VLC tables is use of the same codewords for both the negative and positive value of the variable represented. This can be accommodated by denoting in the codewords the position of the sign bit, using the letter ‘s’. Typically, the sign change affects only one column (at least this is the case with existing MPEG VLC tables). Hence we only need to denote the column that is affected by the sign bit. This is done as shown below:

```
map(vlc, uint, int) table_with_signs sign=2 {
        000         1     5,
        0000.s      2     7,
        0001.00s    3,    -10,
        0001.010    4,    4
};
```

The above specification says that the sign bit (1 for positive, 0 for negative) affects column 2 of the values. Hence 0000.1 results in a value of 7, while 0001.000 results in a value of 10 (-(-10)).

Note that the above specification is quite simple to parse; in fact, a parser for such VLC tables has already been written using Perl, producing VLC definitions in C code and including their complete decoding trees [3]. The parser/code generator also checks on the fly that the VLC table is consistent (no codeword is prefix of another).

4. Composite Types—Objects

4.1 Context-Free Objects

Equipped with the above definitions for fundamental types, we can now examine the definition of composite types or objects. A very useful feature is to be able to immediately identify the type of object that we are dealing with; object identifiers are then a particularly attractive feature. In several cases, the desire for bit efficiency precludes their use (this is the case in MPEG-2 below the slice level). The definition of a composite object can then be expressed as:

```
class object_name {
        element_1;
        [element_2; …]
} [= object_id];
```

The different elements are definitions of elementary bitstream components as we saw in Section 3 (this is half of the truth, as will be explained later). The (optional) *object_id* is a variable length codeword that is specified by the designer of the structure. Clearly, object IDs are intended to be unique within the name space of the decoder[2]. An example of the above definition is given below.

---

[2] Using the notion of a ‘package’ (which can be equivalent to even a tool), we can ensure that each syntax designer has a sufficiently large private name space by concatenating package IDs with object IDs (e.g., audio objects vs. video objects).

```
class picture_header {
        uint(2) picture_coding_type;
        uint(5) temporal_reference;
} =0x000000DD; // whatever the picture start code is
```

The variable length object ID is here described using hexadecimal notation, which by default maps each digit to exactly 4 bits.

The order of declaration of bitstream components is important: it is the same order in which the elements appear in the bitstream. An exception is the object ID, which is specified at the end of the object but appears as its very first entity.

Use of an object ID allows us to substitute objects at the decoder, as we are going to briefly discuss later on. Objects with no ID cannot be substituted, since there is no way to refer to them (unless a textual name space is maintained at the decoder—TBD, but highly unlikely). *[We can get away at the slice level in MPEG-2 if we use only the initial fixed bits of the slice start codes, if they guarantee uniqueness (I don't have the IS handy). This means that, conceivably, we could choose to encode a particular slice using a different tool than the traditional MC-DCT. Further operation of the MPEG-2 decoder would not be affected. Alternatively, we can put don't cares ('X') on the object ID to denote a family of objects. This requires further consideration.]*

We can also encapsulate objects within other objects. In this case, the *element* mentioned at the beginning of this section is an object itself.

4.2 Context-Sensitive Objects

The construction as has been described up to now allows us to define 'objects' of a quite rigid structure: the number of elements included is fixed. In practice, the presense of some elements may be determined by the presense of others. In other words, *context* becomes important. To accommodate this behavior we have to equip our syntactic representation framework with conditional declarations. This conditional behavior is described using the familiar 'if then else' structure found in C and C++, including all the relevant logical operations (&&, ||, etc.). The following example illustrates the procedure.

```
class conditional_object {
        uint(3) foo;
        boolean bar_flag;
        if (bar_flag) {
                uint(8) bar;
        }
        uint(32) more_foo;
};
```

Here the presence of the entity 'bar' is determined by the 'bar_flag'. Another example is:

```
class conditional_object {
    uint(3) foo;
    boolean bar_flag;
    if (bar_flag) {
        uint(8) bar;
    } else {
        vlc(some_vlc_table) bar;
    }
    uint(32) more_foo;
};
```

Here we allow two different representations for ‘bar’, depending on the value of ‘bar_flag’. We could equally well have another entity instead of the second version (the variable length one) of ‘bar’ (another object, or another variable). Note that the use of a flag necessitates its declaration before the conditional is encountered.

In the same category of context-sensitive objects we have the so-called repetitive objects. These simply imply the repetitive use of the same syntax to parse the bitstream, until some condition is met (it is the conditional repetition that implies context, but fixed repetitions are obviously treated the same way). The familiar structures of ‘for’, ‘while’, and ‘do’ loops can be used for this purpose. These types of objects are the most tricky to deal with. When the number of repetitions can be determined by other variables, the situation is quite easy to handle. In many (if not most) cases, however, we do not know that we must stop parsing until we actually read the beginning of the following object (e.g., at the end of a slice we stop because we detect a slice start code). The distinction can be eliminated by considering ‘probing’ and ‘getting’ bits, but this unnecessarily complicates the syntax. The same holds for operations like ‘undo’ and so on: they require test variables that clutter the syntax due to a syntactic inefficiency.

A more appealing solution is to rely, when possible, on the syntax itself. Consider the scanning of macroblocks within a slice: it is terminated when the start code of the next slice is detected (more accurately, when any start code is detected). We must somehow indicate that a particular object may be repeated many times, and when should this repetitive parsing stop. A convenient way of indicating repetition are the ‘*’ and ‘+’ operators used in regular expressions. The notation ‘A*’ means that A appears 0 or more times, while the notation ‘A+’ means that A appears 1 or more times (there is no confusion with arithmetic operators, since these are postfix unary operators). But when does repetition end? When the parser detects the presence of an object ID in the bitstream. This adds the expense of having to search in at least the local object namespace, but this is really a desirable feature: for truly object-oriented video (and graphics) representation, we cannot always predict the sequence with which objects appear in the bitstream (in contrast with the MPEG-1 and MPEG-2 syntaxes that have fully predictable ordering of their ‘objects’).

Although elegant, this technique still has one unresolved problem: in some cases we may want to specify that a particular object should follow the current one. This can be specified by placing the object’s type in brackets after the repetition operator: ‘A+[B]’ indicates that A appears at least once, immediately followed by B. Clearly B must have an object ID in its definition for this idiom to be acceptable. Alternatively, instead of an object B we can directly place the bit string that, when present, should terminate the parsing.

This approach can be used for individual elements, and not only composite objects (or classes). An example is repetitive parsing of VLC coefficients until a specific codeword is detected. This is the case when parsing AC DCT coefficients, a process that is terminated when the end of block (EOB) codeword is detected. With the above construction this problem is solved in a straightforward way with no further syntactic complications. We simply specify:

```
vlc(ac_vlc_table) ac_coeff+[011]
```

This says that 'ac_coeff' is parsed using the VLC table 'ac_vlc_table'. There is at least one 'ac_coeff', and parsing should stop when the bit string '011' is detected in the bitstream. The syntax is still unambiguous: bit strings are never confused with composite object (or class) names, since they can never begin with a digit.

5. Syntactic Description of Complete Structures

Equipped with the preceding definitions, we can now describe the syntax of arbitrary bitstreams of any complexity. Although all of the examples used MPEG-2 video (that's the one with which the author is most familiar with), we expect no complications from the audio or system's specifications. As a proof-of-concept experiment, these syntaxes (including NBC audio) will be recast using the above syntactic description tools and described in future contributions as concrete examples of the application of the proposed framework.

The only missing item in the construction of a large collection of interoperating objects is the designation of the initial object, i.e., the very first one appearing in the bitstream (if it is fixed). This can be easily specified by preceding the 'start' codeword in the definition of the relevant class. If the structure of the encoding is such that the first object can be arbitrary, no such specification is needed: the decoder will find the appropriate object syntax specification using the object ID namespace.

6. Extensibility

In the manner described in the preceding sections, the proposed framework does not provide any significant benefits over the existing C-like syntax used in the MPEG specifications[3]. In order to provide extensibility at the decoder side, we must attack items 2 and 3 discussed in the Introduction.

In order for a decoder (although the word decoder may be too narrow for MPEG-4) to be able to operate using a new tool, two things must be made available to it:
1. the syntax of the data as it is going to be transmitted, and

[3] A possibly important benefit is that it can simplify experimentation with different syntaxes (as in VMs). Because the specification of the syntax is done using a well-defined and self-sufficient approach, software tools can be written that convert the syntactic specification to C or C++ code. The experimenter would then only have to code the decoding algorithm, without worrying about the details of parsing the bitstream.

2. the algorithms that will process this incoming data.

The first item we refer to as meta-data: using some agreed-upon syntax, we are conveying to the decoder the syntax of the data to be received. This doesn't mean that we need to invent yet another language that will describe the structure of the incoming data. The structure presented in Sections 2–5 is sufficient. Of course, we do not want to transmit the textual representation as is; this is useful for human interpretation but is not particularly bit efficient. It is easy to define a binary mapping of the proposed textual specification that represents an equivalent representation of the syntactic structure of the data. This is similar to the '.class' files used in Java to convey type information to the interpreter. We should also note that it is possible to redefine syntactic structures by using the same object ID. This allows us to transparently replace objects in cases where more efficient (or more flexible) representations or processing algorithms are found. A suitable format for the binary mapping of the proposed SDL will be detailed in a future contribution.

After the structure of the data to be transmitted is made known to the decoder, we must specify how this data is going to be processed in order to produce physically presentable material.. To do so, we need to specify a complete application-specific language that will address the requirements of MPEG-4 applications. For purposes of platform independence, this will necessitate the definition of a "virtual processor" (VP), i.e., a hypothetical decoder that will be able to parse the binary representation of this language. The design of the language and especially the VP is an extremely delicate task: on the one hand a wide range of applications need to be accommodated, while on the other hand we have to deal with issues of real-time performance in cost-conscious implementations.

7. Concluding Remarks

We have presented a concrete syntactic representation framework for the description of the binary representation of audiovisual objects and arbitrary binary information. The approach is based on the existing C-like syntax used in MPEG-1 and MPEG-2, with the additional feature of object-orientation. The proposed scheme allows the complete characterization of the bitstream within a single formal construct, hence allowing its use with automated parsing facilities.

Care has been taken to provide mechanisms that can be helpful for specifying coding idioms that frequently appear in bitstream specifications (VLC tables etc.), hence allowing the representation to be both economical and precise. The proposed language provides a direct bridge from the traditional bitstream specification techniques to a one that satisfies the needs of a fully-fledged MPEG-4 L2 SDL.
The focus of this contribution was on the formal description of the coded bitstream. The precise mechanisms that are necessary to produce a complete MPEG-4 SDL have been identified, and include a binary translation of the proposed syntactic description language together with an application-specific language and its corresponding binary representation and virtual processor.

A major benefit of the proposed approach is that in does not constrain in any way subsequent evolution of the MSDL structure. Its primary contribution is that it deals with the intricacies of describing bit-level quantities that may have variable lengths and complex interdependencies. As soon as these data are parsed, they become available in their uncoded representation. It is possible, then, to deal with them as if they were parts of classes in a C++ program, or of a structure in a C program. This is in recognition of the fact that the binary representation of objects and their semantics are orthogonal, and hence they can be dealt with independently.

References

[1] Cheung Auyeung and Mike Danielsen (Motorola), "An MPEG-4 Object Oriented Syntax Proposal" (unnumbered contribution), July 11, 1994.

[2] Cliff Reader (Samsung), Communication in the MSDL AHG reflector, describing a C++ class structure for MPEG-2, May 3, 1995.

[3] Alexandros Eleftheriadis (Columbia University), "mkvlc: A variable length code table compiler," Technical Report 363-94-10, Center for Telecommunications Research, Columbia University, April 1994.