Flavor: A Formal Language for Audio-Visual Object Representation

Version 1.3, 11/97

Alexandros Eleftheriadis and Yihan Fang
{eleft,fang}@ee.columbia.edu

Department of Electrical Engineering
Columbia University
New York, NY 10027

1. Overview

Flavor is a "syntactic description language" that addresses the need to disengage the definition of the bitstream syntax of audio-visual content from the decoding and/or object rendering tools. This requirement originates from the fact that a given syntax specification may be decoded using different implementations of the relevant algorithms. This certainly has been the central theme in the MPEG-1 and MPEG-2 series of specifications, even though the syntax specification utilized both formal and non-formal techniques (i.e., it included explanatory text without which the definition of the syntax would be incomplete). Conversely, it is possible to utilize a given set of tools with various modifications on the syntax of the bitstream data. Examples include modifications in VLC tables, the coded length of parameters, etc. Utilizing a formal methodology to specify the bitstream syntax facilitates support for these features. Furthermore, it makes syntactic description amenable to software tool manipulation, so that automatic generation of bitstream compliance and other tools can be performed. Flavor is currently being used in the MPEG-4 standardization effort to describe the bitstream syntax of audio, video, as well as synthetic content.

This document describes the syntactic description facilities of Flavor, in the form of formal rules. It directly extends the C-like syntax used in MPEG-1 and MPEG-2 into a well-defined framework that lends itself to object-oriented data representations. In particular, Flavor assumes an object-oriented underlying framework in which bitstream units consist of "classes." It then proceeds to extend the typing system by providing facilities for defining bitstream-level quantities, and how they should be parsed.

We first describe elementary constructs, moving to composite syntactic constructs, arithmetic and logical expressions, and finally address syntactic control flow and built-in functions. Syntactic flow control is needed to take into account context-sensitive data. Several examples are used to clarify the structure. A BNF grammar is provided in Section 10.[1]

2. Elementary Data Types

SDL uses the following elementary syntactic elements:

1. Constant-length direct representation bit fields or Fixed Length Codes — FLCs. These describe the encoded value exactly as it is to be used by the decoder.
2. Variable length direct representation bit fields, or parametric FLCs. These are FLCs for which the actual length is determined by the context of the bitstream (e.g., the value of another parameter).
3. Constant-length indirect representation bit fields. These require an extra lookup into an appropriate table or variable to obtain the desired value or set of values.

[1] A more comprehensive description of the overall MPEG-4 System layer design can be found in in the MSDL Web site at http://www-elec.enst.fr/msdl

4. Variable-length indirect representation bit fields (e.g., Huffman codes).

These are described in more detail below.

2.1 Constant-Length Direct Representation Bit Fields

These can be simply represented as:

Rule E.1: Elementary Data Types

[aligned] *type*[(*length*)] *element_name* [= *value*]; // C++-style comments allowed

The *type* can be any of the following: 'int' for signed integer, 'unsigned int' for unsigned integer, and 'bit' for raw binary data. '*length*' indicates the length of the element in bits, as it is stored in the bitstream. The *value* attribute is only present when the value is fixed (e.g., start codes or object IDs), and it may also indicate a range of values (i.e., '0x01..0xAF'). The *type* and the optional *length* are always present, except if the data is non-parsable, i..e., it is not included in the bitstream. The attribute 'aligned' means that the data is aligned on a byte boundary. As an example, a start code would be represented as:

```
aligned bit(32) picture_start_code=0x00000100;
```

An optional numeric modifier, as in aligned(32), can be used to signify alignment on other than byte boundary. Allowed values are 8, 16, 32, 64, and 128. 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 (by default with the most significant bit first–MSBF).

Note that constants are defined using the 'const' attribute:

```
const int SOME_VALUE=255;     // non-parsable constant
const bit(3) BIT_PATTERN=1; // this is equivalent to the bit string "001"
```

To designate binary values, the '0b' prefix is used, similar to the '0x' prefix for hexadecimal numbers, and a period ('.') can be optionally placed every four digits for readability. Hence 0x0F is equivalent to 0b0000.11111.

2.2 Variable Length Direct Representation Bit Fields

This case is covered by Rule E.1, by allowing the '*length*' field to be a variable included in the bitstream, a non-parsable variable, or an expression involving such variables. For example:

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

2.3 Constant-Length Indirect Representation Bit Fields

Here, 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:

Rule E.2: Maps

map *MapName* (*output_type*) [sign=*col*] {
    index, {*value_1*, … *value_M*},
    …
}

These tables are used to translate or map bits from the bitstream into a set of one or more values. The input type of a map (the *index* specified in the first column) is always 'bit'. The *output_type* entry is either a predefined type or a

defined class (classes are defined in Section 3.1). The map is defined as a set of pairs of such indices and values. Keys are binary string constants while values are *output_type* constants. Values are specified as aggregates surrounded by curly braces, similar to C or C++ structures.

As an example, we have:

```
class YUVblocks {// classes are fully defined later on
   uint Yblocks;
   uint Ublocks;
   uint Vblocks;
}
// a table that relates the chroma format with the
// number of blocks per signal component
map blocks_per_component ( YUVblocks ) {
   0b00,     {4,   1,    1},   // 4:2:0
   0b01,     {4,   2,    2},   // 4:2:2
   0b10,     {4,   4,    4}    // 4:4:4
};
```

The next rule describes the use of such a map by the declaration of a variable of type vlc.

Rule E.3: Mapped Data Types

type (*MapName*) *name*;

Example:

```
YUVblocks(blocks_per_component) chroma_format;
```

Using the above declaration, we can access a particular value of the map using the construct chroma_format.Ublocks.

2.4 Variable Length Indirect Representation Bit Fields

For a variable length element utilizing a Huffman table, an identical specification to the fixed length case is used:

```
class val {
   uint foo;
   int bar;
};

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

The only difference is that the indices of the map are now of variable length. The variable-length codewords are (as before) binary strings, expressed by default in ‘0b’ or ‘0x’ format, optionally using the period (‘.’) every four digits for readability.

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 (or even variable length) representation. To allow for such exceptions, parsable type declarations are allowed for map values.

This is illustrated in the following example (we use the class type ‘val’ as defined above):

```
map sample_map_with_esc ( val )  {
   0b0000.001,     {0,   5},
   0b0000.0001,    {1,   -14},
   0b0000.0000.1,  {5, int(32)},
   0b0000.0000.0,  {0,   -20}
```

```
};
```

As written above, when the codeword 0b0000.0000.1 is encountered in the bitstream, then the value '5' is asigned to the value of the first element (val.foo), while the following 32 bits will parsed and assigned as the value of the second element (val.bar). Note that, in case more than one element utilizes a parsable type declaration, the order is significant and is the order in which elements are parsed. In addition, the type within the map declaration must match the type used in the class declaration associated with the map's return type. Using this construct, the complete behavior of the VLC mapping is described in a concise manner in a single place.

3. Composite Data Types

3.1 Classes

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

Rule C.1: Classes

[aligned] class *object_name* [extends *parent_class*] [: bit(*length*) [*id_name*]= *object_id* | *id_range*] {
    [*element*; …] // zero or more elements
}

The different elements within the curly braces are definitions of elementary bitstream components as we saw in Section 2, or control flow that is discussed later on. The *object_id* is optional, and if present is the key multiplexing entity for individual objects. The *id_range* is specified as *start_id .. end_id*, inclusive of both bounds, to express that the object can have a range of possible id's.

The optional 'extends *parent_class*' specifies that the class is "derived" from another class. Derivation means that all information present in the base class is also present in the derived class, and that all such information *precedes* in the bitstream any additional bitstream syntax declarations that are specified in the new class.

Examples:

```
class slice: aligned bit(32) slice_start_code=0x00000101 .. 0x000001AF {
// here we get vertical_size_extension, if present
   if (scalable_mode==DATA_PARTITIONING) {
      unsigned int(7) priority_breakpoint;
   }
   …
}

class foo {
   int(3) a;
   ...
}

class bar extends foo {
    int(5) b;      // this b is preceded by the 3 bits of a
```

```
   int(10) c;
   ...
}
```

The order of declaration of bitstream components is important: it is the same order in which the elements appear in the bitstream. In the above examples, foo::b immediately precedes foo::c in the bitstream.

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

3.2 Parameter types

A parameter type defines a class with parameters. This is to address cases where the data structure of the class depends on variables of one or more other objects. Because Flavor follows a declarative approach, references to other objects cannot be performed directly (none is instantiated). Parameter types provide placeholders for such references, in the same way as the arguments in a C function declaration. The syntax of a class definition with parameters is as follows.

Rule C.2: Class Parameter Types

[aligned] class *object_name* [(parameter list)] [extends *parent_class*]
[: bit(*length*) [*id_name*]= *object_id* | *id_range*] {
[*element*; …] // zero or more elements
}

The parameter list is a list of type name and variable name pairs separated by commas. Any element of the bitstream, or value derived from the bitstream with a vlc, or a constant can be passed as a parameter.

A class that uses parameter types is dependent on the objects in its parameter list, whether class objects or simple variables. When instantiating such a class into an object, the parameters have to be instantiated objects of their corresponding classes or types.

Example:

```
class A {
  // class body
  ...
  unsigned int(4) format;
}

class B( A a, int i ) {       // B uses parameter types
  unsigned int(i) bar;
  ...
  if( a.format == SOME_FORMAT ) {
    ...
  }
  ...
}

class C {
  int(2) I;
  A a;
  B( a, I) foo; // instantiated parameters are required
}
```

4. Arrays

Arrays are defined in a similar way as in C/C++, i.e., using square brackets. Their length, however, can depend on run-time parameters such as other bitstream values or expressions that involve such values. The array declaration is applicable to both elementary as well as composite objects.

Rule A.1: Arrays

typespec name [*length*];

---

typespec is a type specification (including bitstream representation information, e.g. 'int(2)'), *name* is the name of the array, and *length* is its length. For example, we can have:

```
unsigned int(4) a[5];
int(10) b;
int(2) c[b];
```

Here 'a' is an array of 5 elements, each of which is represented using 4 bits in the bitstream and interpreted as an unsigned integer. In the case of 'c', its length depends on the actual value of 'b'. Multi-dimensional arrays are allowed as well. The parsing order from the bitstream corresponds to scanning the array by incrementing first the right-most index of the array, then the second, and so on .

In several situations, it is desirable to load the values of an array one by one, in order to check for example a terminating or other condition. For this purpose, an extended array declaration is allowed in which *individual* elements of the array may be accessed.

Rule A.2: Partial Arrays

typespec name[[*index*]];

Here *index* is the element of the array that is defined. Several such partial definitions can be given, but they must all agree on the type specification. This notation is also valid for multidimensional arrays. For example:

```
int(4) a[[3]][[5]];
```

indicates the element a(5, 3) of the array, while

```
int(4) a[3][[5]];
```

indicates the entire sixth column of the array, and

```
int(4) a[[3]][5];
```

indicates the entire fourth row of the array, with a length of 5 elements.

Note that 'a[5]' means that the array has five elements, whereas 'a[[5]]' implies that there are at least six.

5. Arithmetic and Logical Expressions

All standard arithmetic and logical operators of C/C++/Java are allowed, including their precedence rules. *Ed.: A standard should be cited, or the precedence rules duplicated.*

6. Non-Parsable 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, non-parsable variables are allowed. These are strictly of local scope to the class they are defined in. They can be used in expressions and conditions in the same way as bitstream-level variables. In the following example, the number of non-zero elements of an array is computed.

```
unsigned int(6) size;
int(4) array[size];
```

```
…
int i; // this is a temporary, non-parsable variable
for (i=0, n=0; i<size; i++) {
  if (array[i]!=0) n++;
}

int(3) coefficients[n];
// read as many coefficients as there are non-zero elements in array
```

7. Syntactic Flow Control

The syntactic flow control provides constructs that allow conditional parsing, depending on context, as well as repetitive parsing. The familiar C/C++ if-then-else construct is used for testing conditions. Similarly to C/C++/Java, we consider zero as corresponding to false, and non-zero as correspond to true.

Rule FC.1: Flow Control Using If-Then-Else

if (*condition*) {
…
} [else if (*condition*) {
…
}] [else {
…
}]

The following example illustrates the procedure.

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

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

```
class conditional_object {
  unsigned int(3) foo;
  bit(1) bar_flag;
  if (bar_flag) {
    unsigned int(8) bar;
  } else {
    unsigned int(some_vlc_table) bar;
  }
  unsigned int(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. Also, if a variable appears twice (as in the example above), the types should be identical.

In order to facilitate cascades of if-then-else constructs, the ‘switch’ statement is also allowed.

Rule FC.2: Flow Control Using Switch

switch (*condition*) {

[case *label1*: …]
[default:]
}

The same category of context-sensitive objects also includes iterative definitions of 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.

Rule FC.3: Flow Control Using For

for (*expression1*; *expression2*; *expression3*) {
…
}

expression1 is executed prior to starting the repetitions. Then *expression2* is evaluated, and if it is non-zero (true) the declarations within the braces are executed, followed by the execution of *expression3*. The process repeats until *expression2* evaluates to zero (false).

Rule FC.4: Flow Control Using Do

do {
…
} while (*condition*);

Here the block of statements is executed until *condition* evaluates to false. Note that the block will be executed at least once.

Rule FC.5: Flow Control Using While

while (*condition*) {
…
}

The block is executed zero or more times, as long as *condition* evalutes to non-zero (true).

To reduce the need for temporary variables (counters etc.), a 'repeat' statement is also defined.

Rule FC.6: Flow Control Using Repeat

repeat (*parameter*) {
…
}

The block within the braces will be repeated as many times as indicated by *parameter*. Note that *parameter* is only examined once, at the beginning of the iterations.

8. Bitstream Tests

In several instances, the syntactic description of the bitstream requires a look-ahead into bits that follow the currently parsed element. For this reason, two constructs are provided.

Rule B.1: Non-Destructive and Destructive Bitstream Tests

[*bitstring*] or [*bitstring**]

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 contruct '[*bitstring**]' performs the same operation, but if the string is found, the bits are removed from the bitstream.

Example:

```
do {
   int(ac_vlc_table) ac_coeff;
} while (! [0b011]);
```

This says that 'ac_coeff' is parsed using the 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.

Similarly, it is sometimes desirable to perform these tests in a byte-aligned fashion. In this case, the test is not performed at the current bitstream position, but at the immediately following boundary.

Rule B.1: Non-Destructive and Destructive Bitstream Tests with Alignment
[aligned(*size*) *bitstring*] or [aligned(*size*) *bitstring**]

As before, the argument to 'aligned' can be 8, 16, 32, 64, or 128. If the argument is omitted, the value 8 is assumed (byte alignment).

9. Built-in Functions

Built-in functions are used to reduce the amount of expressions that are found in the Flavor description, and improve readibility, simplicity, as well as performance (they can implemented intelligently). The inclusion of the definition (where applicable) does not specify any particular preferred implementation, and it is only intended to unambiguously demonstrate the operation of the function. The standard built-in functions are indicated below.

```
// length of parameter in native implementation after parsing
int sizeof(parameter)

// length of a parameter in bits as described in its syntax
int bitsizeof(parameter)
```

Ed.: The above list is tentative; further built-in functions will be defined as the descriptions of the various WD/VMs are finalized.

10. Flavor Grammar

Ed.: The new BNF grammar will be added in the Feb 97 meeting.