Audio Engineering Society
Convention Paper 9537

Presented at the 140th Convention
2016 June 4–7 Paris, France

Development tools for modern audio codecs

Jonas Larsen[1] and Martin Wolters[2]

[1] Dolby Germany GmbH, Deutschherrnstrasse 15 – 19, Nuremberg, 90429, Germany
jonas.larsen@dolby.com

[2] Dolby Germany GmbH, Deutschherrnstrasse 15 – 19, Nuremberg, 90429, Germany
martin.wolters@dolby.com

ABSTRACT

The Dolby Bitstream Syntax Description Language (BSDL) is a generic, XML-based language for describing the syntactical structure of compressed audio-visual streams. This paper describes how the representation of a bitstream syntax in the BSDL is used to ease the development of serialization, deserialization and editing tools. Additionally, the formal syntax description allows realizing a range of novel analysis methods including bitstream syntax coverage measurements, detailed bitrate profiles, and the automatic generation of rich specification documentation. The approach is exemplified using the AC-4 codec.

1. INTRODUCTION

Traditionally, the bitstream syntax of a codec is described in a loosely defined format using pseudo-C syntax or a tabular format. This is still the most common format for International Standards such as ETSI or ISO. Formal bitstream syntax description languages include the MPEG-21 Bitstream Syntax Description Language [1] and MSDL (MPEG-4 Syntactic Description Language, [2]) / Flavor (Formal Language for Audio-Visual Object Representation, [3]). They provide an unambiguous and machine-readable specification of a codec's bitstream syntax. A more elaborate overview can be found in [4].

State-of-the-art audio codecs such as AC-4 [5][6] support what is often referred to as "NGA" (Next Generation Audio) which commonly refers to immersive, personalized, and object-based audio [7]. While it was common to just transmit audio as simple mono, stereo, or 5.1 audio representations, NGA systems expect a much broader and more flexible audio system: Channel-based audio representations are extended to include channel configurations such as 5.1.2, 5.1.4, or 7.1.4 (where the last digit refers to the number of speakers above the listener). NGA systems also offer several system level configurations to support advanced use cases such as personalization, object-based audio content and loudness management. Last but not least, the requirement for achieving good quality at low datarates leads to a larger number of coding tools.

A first indication of the increased complexity of such audio codec standards is the necessary amount of documentation. Comparing the specifications of Enhanced AC-3 [8] as a common legacy codec and AC-4, a state-of-the-art codec designed to specifically address the requirements of NGA systems, leads to the following results:

Enhanced AC-3	
ETSI TS 102 366, Chapter 4	21 pp
ETSI TS 102 366, Annex E	31 pp
Total	**52 pp**

AC-4	
ETSI TS 103 190, chapter 4	100 pp
ETSI TS 103 190-2, chapter 6	75 pp
Total	**175 pp**

Table 1: Comparing the complexity of Enhanced AC-3 and AC-4 by the number of pages used to describe their respective syntax elements.

Consequently managing bitstream syntax and verifying codec implementations becomes an increasingly complex task. Without the use of a formal bitstream syntax description language and its corresponding automatically generated tools, it is an overwhelming task to develop robust implementations.

The Dolby Bitstream Syntax Description Language (referred to as "BSDL" from here) and its supporting tools are described in this publication, using AC-4 as an example, and parallels are drawn to other bitstream syntax description languages and their implementations. We describe how the BSDL and its supporting tools were used during the development of AC-4 and discuss the advantages as well as limitations of the approach.

2. LANGUAGE OVERVIEW

The BSDL describes the bitstream structure from the highest level down to the lowest level of granularity. The BSDL is XML-based, i.e., the specification of a particular bitstream syntax is an XML document, named a Bitstream Syntax Description (BSD). See Table 2 and Table 3 for an example.

```
<syntax>
  <function name='ac4_syncframe'>
    <uint name='sync_word' number='16' />
    <error condition='sync_word != 0xAC40 &&
sync_word != 0xAC41' text='Wrong sync_word'
    />
    <call name='frame_size' />
    <call name='raw_ac4_frame' />
    <if condition='sync_word == 0xAC41'>
      <uint name='crc_word' number='16' />
    </if>
  </function>
  <function name='frame_size'>
    <uint name='frame_size' number='16' />
    <if condition='frame_size == 0xffff'>
      <uint name='frame_size' number='24' />
    </if>
  </function>
```

Table 3: Excerpt of the AC-4 BSD.

```
ac4_syncframe()
{
    sync_word;                          16 bits
    if (sync_word != 0xAC40 && sync_word !=
0xAC41) {
        ERROR: Wrong sync_word
    }
    frame_size();
    raw_ac4_frame();
    if (sync_word == 0xAC41) {
        crc_word;                       16 bits
    }
} /* ac4_syncframe() */

frame_size()
{
    frame_size;                         16 bits
    if (frame_size == 0xffff) {
        frame_size;                     24 bits
    }
} /* frame_size() */
```

Table 2: The source code from Table 2 reformatted to using a pseudo-C syntax.

The building blocks of the BSDL are:

- Fixed-sized and variably-sized (used for entropy coded data) bit fields
- Byte alignment
- Control structures (if-, switch-, while- and for-statements)

- Definition and reference to compound structures (similar to "class" in Flavor)
- Helper variables, which are multi-dimensional integer arrays
- Assignments and expressions, which support arithmetic, bitwise and logic operators according to the common subset of C- and Python-syntax
- Error detection conditions

Variables are dynamically created when they are initially assigned a value, i.e., unlike in C it is not necessary to declare or define variables before their usage. Array elements before the addressed element are implicitly assigned a default value (zero for integer elements, the empty array for array elements). For example, if the variable x does not already exist, the assignment "x[2][3] = 4" results in x being a two-dimensional array with value [[], [], [0, 0, 0, 4]].

2.1. Text output

Similar to the Flavor package, which supports C++ and Java, serialization and deserialization code is automatically generated from the BSD, using a BSDL-to-python compiler or a BSDL-to-C compiler, as summarized in Figure 1.

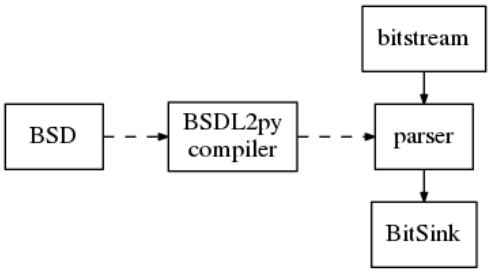


Figure 1: Overview of the BSDL compilation process

At compile-time (dashed lines), the BSD is converted to executable python- or C-code. At runtime (solid lines), the bitstream is deserialized by the parser. The output of the parser is consumed by a generic bit sink component that receives the bitstream data along with syntactical metadata (for fixed-width elements, the element's name and width, and the current call stack). As an example, a specific bit sink may echo the content of the bitstream:

```
ac4_syncframe():
    sync_word: 44096 (0xac40)
    frame_size():
        frame_size: 683 (0x2ab)
    raw_ac4_frame():
        ac4_toc():
            bitstream_version: 0 (0x0)
            sequence_counter: 0 (0x0)
            b_wait_frames: 1 (0x1)
```

Table 4: Displaying a trace of the data in a particular AC-4 bitstream. The corresponding BSD is defined in Table 2.

2.2. Converting bitstreams to or from XML

Similar to XFlavor, a bitstream can be converted to an XML format. This is achieved by using a bit sink that reformats the bitstream data as an XML document:

```
<Frames>
   <ac4_syncframe>
      <sync_word>44096</sync_word>
      <frame_size>
         <frame_size>683</frame_size>
      </frame_size>
      <raw_ac4_frame>
         <ac4_toc>
            <bitstream_version>0</bitstream_version>
            <sequence_counter>0</sequence_counter>
            <b_wait_frames>1</b_wait_frames>
```

Table 5: Tracing the data in a particular AC-4 bitstream. The corresponding BSD is defined in Table 2.

The input to the parser is also a generic bit source, and the XML-formatted bitstream can be converted to binary format by combining a bit source that reads from an XML document with the bit sink that serializes the data to a bitstream.

The possibility of converting bitstreams to XML and converting XML to bitstreams allows for making arbitrary changes to bitstreams subject only to the constraints of the BSD. Such modifications could range from changing single bits to much more complex modifications like injecting metadata into an existing bitstream. This enables the testing of codec features that

```
ac4_syncframe():                        ac4_syncframe();
                                        ac4_syncframe();
                                        {
    sync_word: 44096 (0xac40)               sync_word;                          16 bits
                                            if (sync_word != 0xAC40 and sync_word != 0xAC41)
                                                ERROR: Wrong sync_word
    frame_size():                           frame_size();
                                            frame_size()
                                            {
        frame_size: 683 (0x2ab)                 frame_size;                     16 bits
                                                if (frame_size == 0xffff) {
                                                    }
                                                } /* frame_size() */

    raw_ac4_frame():                            raw_ac4_frame();
                                                raw_ac4_frame()
                                                {
        ac4_toc():                                  ac4_toc();
                                                    ac4_toc()
                                                    {
            bitstream_version: 0 (0x0)                  bitstream_version;      2 bits
                                                        if (bitstream_version == 3) {
                                                        }
            sequence_counter: 0 (0x0)                   sequence_counter;       10 bits
            b_wait_frames: 1 (0x1)                      b_wait_frames;          1 bit
```

Table 6: Bitstream trace (1st column) annotated with the corresponding bitstream syntax (2nd column).

are not natively supported by the initial encoding implementation.
Some differences between MSDL / Flavor / XFlavor and the BSDL and its implementations are

- The deserialized data is not stored as an in-memory object, but routed directly through the parser from the bit source to the bit sink.

- Because syntactical metadata is provided to the bit sink (and because both bit source and bit sink are generic components, which can be chosen at runtime), multiple applications (serialization, deserialization, tracing, binary-to-XML conversion, XML-to-binary conversion) can be supported without the need of changing or re-configuring the compiler.

- No XML schema file is created for the XML-formatted bitstream. A schema file is not necessary for reading the XML-formatted bitstream, because its structure is already constrained and described by the BSD.

2.3. BSDL interpreter

Unlike other tools, BSDL can also be interpreted. The BSDL interpreter operates simultaneously on a BSD and a bit source (see Figure 2). In addition to the bitstream data, the bit sink is provided with the full BSD source code. This enables an "annotated" tracing mode, where the values in a bitstream are annotated

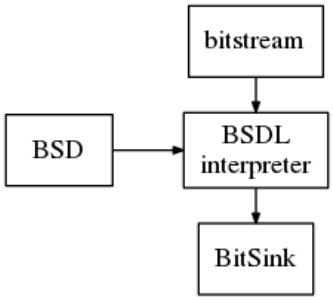


Figure 2: Overview of the BSDL interpreter.

with their corresponding bitstream syntax definition, see Table 6. The simultaneous knowledge of the bitstream data along with its syntax definition (BSDL source code) is reminiscent of using a debugger with a conventional programming language.

2.4. Bitstream Syntax Documentation

During the AC-4 standardization process, the syntactical description of the specification document [5] was derived from the same BSD that served as the basis for multiple development tools. This helped enforcing consistency even in the earliest stages of developing the codec, when the bitstream syntax was still undergoing changes.

3. USING BSDL TO MEASURE BITSTREAM SYNTAX COVERAGE OF TEST VECTORS

Testing and validating software implementations such as the implementation of an audio decoder has become an increasingly difficult and extensive task. In addition, it is basically impossible to test all conditions in which an audio decoder may be used in the field since the specific input signals and sequences are unknown during testing. In order to decide whether an implementation was tested sufficiently, a number of quantitative measurements to describe test coverage have been proposed (excerpt from [9]):

- Line coverage
- Statement coverage
- Branch coverage
- N-length sub-path coverage
- Path coverage
- Multicondition or predicate coverage
- Trigger every assertion check in the program
- Loop coverage
- Fuzzy decision coverage
- Relational coverage

In addition to these more generic metrics, we are using the so called "bitstream syntax coverage" to specifically evaluate the test coverage for an audio decoder.

3.1. BSDL source code coverage

In analogy with C and MATLAB being programming languages (whose source files conventionally carry the suffix .c or .m, respectively), the BSDL is a programming language, and its source files are the BSDs, which are also XML documents.

For a program implemented in a conventional programming language, a source code coverage tool (example: gcov for C / C++) can be used for measuring which parts of the program's source code were executed when the program was executed with a given set of input stimuli.

Similarly, a source code coverage tool for the BSDL analyzes the coverage of a BSD from a given set of input bitstreams that conform to the BSD. The result is a report of the line-by-line coverage of the BSD, including the ranges of any fixed- and variably-sized elements that were present in the input streams.

Syntax	No. of bits

raw_frame()
{
  number1;
  number2;
  if (number1 == 0 && number2 == 0) {
  number3 = 3;
  } else {
  number3;
  }
}
	
	1
	1
	
	1

Table 7: Simple example of a bitstream syntax definition, formatted using pseudo-C syntax.

Sequence	Bits	No. of frames
A	010	1
B	101	1
C	00	1
D	111011110100	4

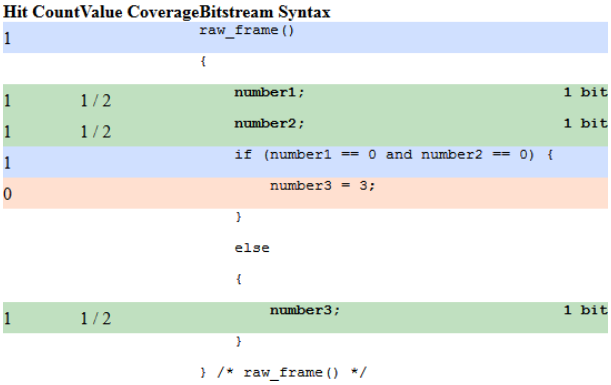
Table 8: Example test vectors.

The example BSD in Table 7 and test vectors in Table 8 lead to the following syntax coverage:

Sequence A covers all syntax bits (*number1*, *number2* and *number3*) but not all possible values. The combination of sequences A and B covers all bits and all values that can be explicitly transmitted. The combination of sequences A, B, and C additionally covers all possible branches and thus all possible values of the syntax bits. The combination of sequences A, B, C, and D covers all seven possible frames, but frame repetitions are not covered and also the order in which frames are to be decoded only covers one possible

configuration. Provided that the decoder algorithm does not only depend on the current frame but also on previous frames (which is the case in all modern audio codecs), such dependencies can, for example, be detected through code coverage measurements.

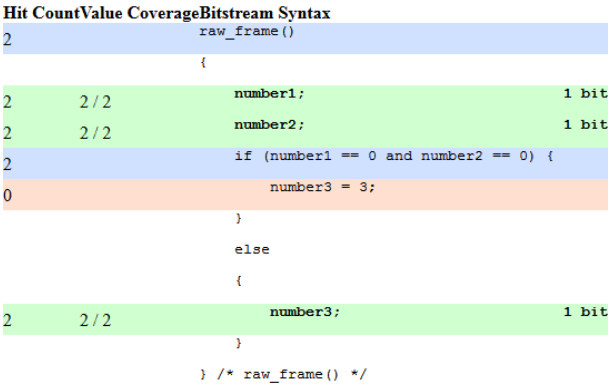
This analysis is automatically performed by the BSDL coverage tool. Using color codes the following bitstream syntax coverage reports highlight the syntax elements for which all possible values are covered (light green), syntax elements for which at least one value is covered (dark green), syntax elements that are not covered (not shown), syntax code that is covered (blue) and syntax code that is not covered (light red). The report also shows how often each line is covered, and how many values have been covered for a specific element. The example in Table 8 leads to the following coverage reports:



Hit Count	Value Coverage	Bitstream Syntax	
1		raw_frame()	
		{	
1	1 / 2	number1;	1 bit
1	1 / 2	number2;	1 bit
1		if (number1 == 0 and number2 == 0) {	
0		number3 = 3;	
		}	
		else	
		{	
1	1 / 2	number3;	1 bit
		}	
		} /* raw_frame() */	

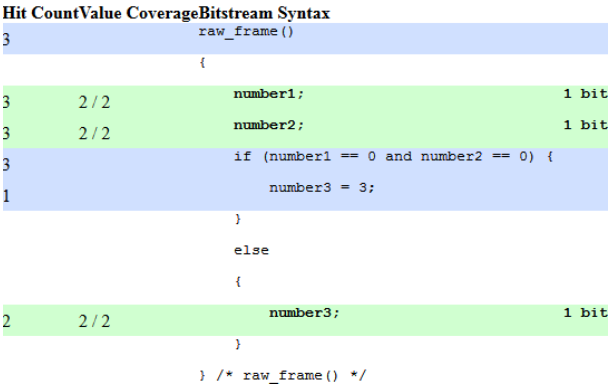
Figure 3: Coverage after sequence A.

Notice that no coverage difference is reported between A + B + C and A + B + C + D. Since it is impossible in modern audio codecs to cover all possible combinations within a single frame, we decided not to track which combinations of values have been covered by a given test set.



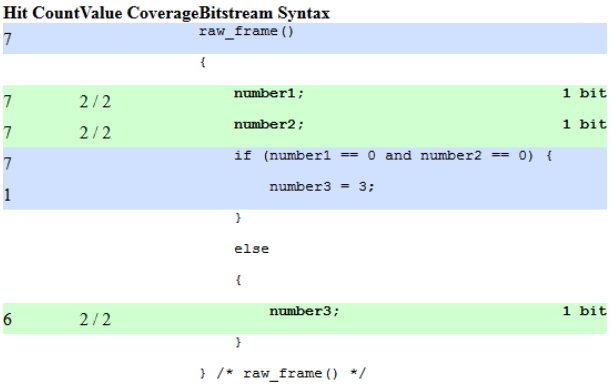
Hit Count	Value Coverage	Bitstream Syntax	
2		raw_frame()	
		{	
2	2 / 2	number1;	1 bit
2	2 / 2	number2;	1 bit
2		if (number1 == 0 and number2 == 0) {	
0		number3 = 3;	
		}	
		else	
		{	
2	2 / 2	number3;	1 bit
		}	
		} /* raw_frame() */	

Figure 4: Coverage after sequences A + B.



Hit Count	Value Coverage	Bitstream Syntax	
3		raw_frame()	
		{	
3	2 / 2	number1;	1 bit
3	2 / 2	number2;	1 bit
3		if (number1 == 0 and number2 == 0) {	
1		number3 = 3;	
		}	
		else	
		{	
2	2 / 2	number3;	1 bit
		}	
		} /* raw_frame() */	

Figure 5: Coverage after sequences A + B + C.



Hit Count	Value Coverage	Bitstream Syntax	
7		raw_frame()	
		{	
7	2 / 2	number1;	1 bit
7	2 / 2	number2;	1 bit
7		if (number1 == 0 and number2 == 0) {	
1		number3 = 3;	
		}	
		else	
		{	
6	2 / 2	number3;	1 bit
		}	
		} /* raw_frame() */	

Figure 6: Coverage after sequences A + B + C + D.

It is also common that specific audio decoder implementations targeting a certain use case usually support only a subset of elements described in a standard, usually in order to limit the complexity of such implementation. Such subsets are defined in AC-4 as so called "levels". The test set for a given decoder implementation is dependent on the targeted decoder level. The BSDL coverage tool therefore allows to reduce the number of syntax elements taken into account when measuring the bitstream syntax coverage. This is achieved by adding a syntax element to a whitelist. An element can either be excluded from the coverage measurement in general, or only certain values of the element can be excluded. For the latter, a good example is the sync word. (See for example Table 2.) The sync word is designed to allow for a robust detection of a frame start. It is 16 bits wide but only two values are considered valid. All other values are therefore excluded from coverage measurements through a corresponding whitelist.

The BSDL coverage tool was used to estimate, and iteratively extend, the completeness of the reference collection of AC-4 test bitstreams for a level-3 decoder implementation. The resulting bitstream syntax coverage statistics were:

- 100 % syntax bits coverage
- > 90 % syntax code coverage
- Full range coverage of > 50 % of the syntax elements

In addition to various other coverage metrics such as requirements coverage, code coverage, use-case coverage and coverage on various processors and operating systems, this provided additional confidence that the reference implementation is robust enough to support not only today's systems but also future applications.

4. BITRATE PROFILING

For conventional programming languages, profiling tools (example: gprof for C / C++) measure, for example, the time or memory that different sections of a program consume during execution. Advanced profilers are capable of measuring different quantities than time, such as cache misses or other hardware-events.

```
32.769 kbps (100.00 %) - ac4_syncframe
31.457 kbps (95.99 %) - raw_ac4_frame
28.349 kbps (86.51 %) - ac4_substream
 2.438 kbps (7.44 %) - ac4_toc
 0.938 kbps (2.86 %) - frame_size
 0.844 kbps (2.57 %) - ac4_presentation_v2
 0.670 kbps (2.04 %) - ac4_presentation_su
 0.563 kbps (1.72 %) - substream_index_tab
 0.422 kbps (1.29 %) - emdf_info
 0.358 kbps (1.09 %) - metadata
 0.281 kbps (0.86 %) - emdf_protection
 0.281 kbps (0.86 %) - ac4_substream_group
 0.200 kbps (0.61 %) - drc_frame
 0.164 kbps (0.50 %) - ac4_substream_info_
 0.106 kbps (0.32 %) - drc_config
 0.100 kbps (0.31 %) - drc_decoder_mode_co
 0.094 kbps (0.29 %) - ac4_presentation_su
 0.076 kbps (0.23 %) - drc_compression_cur
 0.070 kbps (0.21 %) - extended_metadata
 0.070 kbps (0.21 %) - ac4_sgi_specifier
 0.070 kbps (0.21 %) - drc_data
 0.053 kbps (0.16 %) - dialog_enhancement
```

Table 9: Flat bitrate profile for stereo AC-4 at 32.8 kbps.

In the same manner, BSDs can be subjected to profiling. The BSDL profiler measures neither time nor memory, but the number of bits that are consumed in each section of the BSD, for a given input bitstream. The measurement results in a flat profile and a call-graph profile, which can be used to analyze and tune the efficiency of the compression that the codec provides. For AC-4, bitrate profiles were also used to compare the bitrate of metadata versus audio payload under various operation modes (see Table 9 and Figure 7). The implementations of the BSDL profiler and BSDL coverage tool are both based on the architecture in 1 with bit sink components that maintain a running bit count (profiler) or coverage information (coverage tool) for each syntax element.

5. CONCLUSION

The feature set and complexity of NGA codecs has significantly increased, as compared to the current- or previous-generation codecs. In this article we reviewed how the BSDL and derived tools were used during the development of AC-4. We applied the familiar concepts of source code coverage and profiling to the BSDL which resulted in novel development tools, that helped to analyze and ensure the quality of the implementation of the codec.

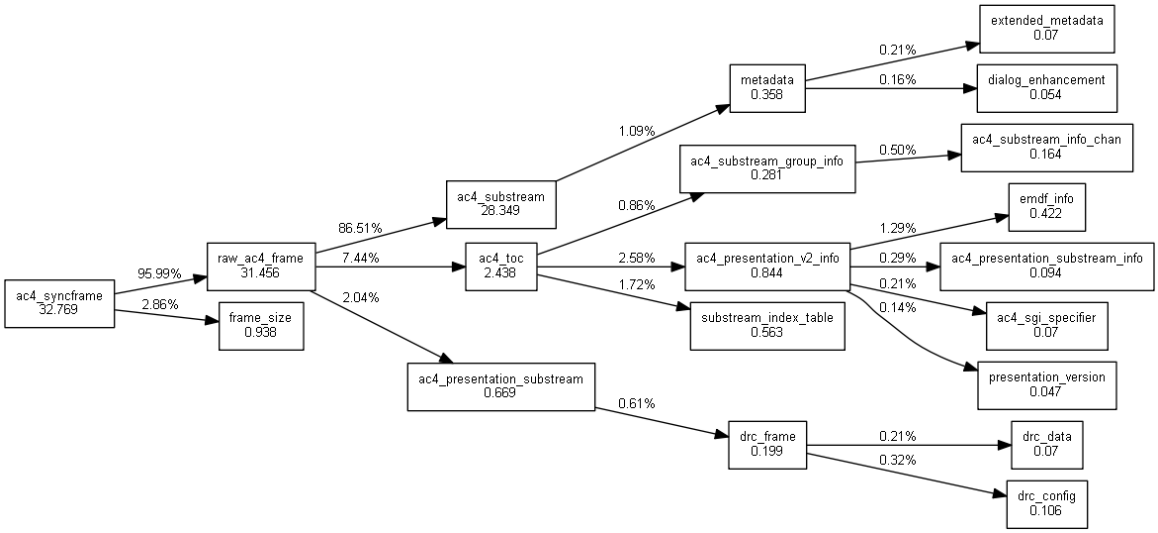


Figure 7: Hierarchical profile generated from the AC-4 BSD and a stereo bitstream at 32.8 kbps. For each subtree, the bitrate is indicated in units of kbps and as a percentages of the total bitrate.

6. REFERENCES

[1] ISO/IEC 21000-7:2004 Information technology -- Multimedia Framework (MPEG-21) -- Part 7: Digital Item Adaptation, 2004

[2] ISO/IEC 14496-1:2010 Information technology -- Coding of audio-visual objects -- Part 1: Systems

[3] http://flavor.sourceforge.net/ (last visited 2016-03-13)

[4] Hyungyu Kim et.al., "Overview of Bitstream Syntax and Parser Description Languages for Media Codecs", IEEK Transactions on Smart Processing and Computing, vol. 2, no. 3, June 2013

[5] ETSI TS 103 190 V1.1.1 "Digital Audio Compression (AC-4) Standard" and ETSI TS 103 190-2 V1.1.1 "Digital Audio Compression (AC-4) Standard, Part 2: Immersive and personalized audio"

[6] Kjörling K. et.al. , "AC-4 – The Next Generation Audio Codec", Audio Engineering Society Convention Paper Presented at the 140th Convention 2016 June 4–7 Paris, France.

[7] Most documentation around NGA is currently not yet publicly available outside of related standards organizations; one overview can be found here: http://www.theiabm.org/news/blog/blog_detail.dvb-issues-questionnaire-on-next-generation-audio-requirements.html (last visited 2016-03-13)

[8] ETSI TS 102 366 V1.2.1 "Digital Audio Compression (AC-3, Enhanced AC-3) Standard"

[9] Cem Kaner, "Software Negligence & Testing Coverage", STAR Conference, Orlando, 1996