DEPARTMENT OF ELECTRICAL ENGINEERING
TECHNICAL REPORT

XFLAVOR: THE NEXT GENERATION OF MEDIA REPRESENTATION

Danny Hong and Alexandros Eleftheriadis

CU/EE/TR 2002-05-022

Columbia University
Department of Electrical Engineering
500 W. 120th Street, MC 4712
New York, NY 10027

http://www.ee.columbia.edu

XFlavor: The Next Generation of Media Representation

Danny Hong
Columbia University
500 West 120th Street, Room 1312
New York, NY 10027
01-212-939-7154
danny@ee.columbia.edu

Alexandros Eleftheriadis
Columbia University
500 West 120th Street, Room 1312
New York, NY 10027
01-212-854-8670
eleft@ee.columbia.edu

ABSTRACT
Flavor has been created as a framework for describing compressed multimedia bitstreams in a formal way so that the bitstream parsing/generating code can be automatically generated. For this, Flavor comes with a software tool that translates Flavor descriptions into C++ or Java code. In this paper, we describe XFlavor, an extended version of the translator, which provides XML features. XFlavor has the capability to transform a Flavor description into XML schema and it can also produce code for generating XML documents corresponding to the bitstreams described by Flavor. As part of XFlavor, a compression tool for converting the XML representation back into the original bitstream format is discussed as well.

Categories and Subject Descriptors
E.2 [**Data**]: Object Representation; H.5.1 [**Information Interfaces and Presentation**]: Multimedia Information Systems; D.3.4 [**Programming Languages**]: Processors—*code generation, compilers, parsing*

General Terms
Design, Languages

Keywords
Media Representation, Flavor, XFlavor, XML, XML Schema, XSLT, XML Compression

1. INTRODUCTION
Flavor [1, 2, 3] is a framework developed as the means to describe compressed multimedia data with the purpose of simplifying and speeding up the development of software that processes such data. From the Flavor description, the bitstream parsing and generating code is automatically generated with a well-defined interface. However, the software still interacts with the data in the bitstream level and this poses limitations in terms of flexibility and extensibility. For example, adding or deleting extra information in the original data would be almost impossible, as it would most likely destroy the original bitstream syntax of the data. This causes the data to be incompatible with the existing, specification compliant decoders. As a solution, we need a way to abstract the biststream layer and enable applications to solely work with the semantic values of the bits. Hence, as long as all the necessary semantic values are availabe in the data, applications should be able to process the data regardless of the structure it may take.

With a growing number of multimedia applications and their popularity, it's necessary to devise a representation that provides much more than just compression, including interoperability among different applications. XML [4] defines the structure of data for easier processing by different applications. The structure, however, is not fixed, allowing modification of the data without breaking the interoperability. For example, the data can be customized to an application by adding extra information for the application specific use. This extra information, though it modifies the structure of the data, does not impede other applications from using the original part of the data that is still available. In an XML document, each element includes the context information of its content and the information does not get changed when adding/deleting extra elements. Thus, regardless of the structure of the data, due to the self-describing nature of the data, correct semantic values can still be obtained. Unfortunately, this is not the case with the bitstream-represented data. Such as, in a GIF [5] image, adding extra information anywhere in the bitstream (other than at the end of the bitstream) renders the bitstream useless. Additionally, the extra information cannot be easily distinguished from the bitstream.

Along with the growing number of multimedia applications, we also see a tremendous growth in the number of multimedia content that are available. As a result, searching and finding the right content have become a very difficult process. XML also provides a solution in regards to this problem. XML enables data to be self-describing (via the use of tags, attributes, and the structure of elements), and this, in essence, allows easy client-side processing of the data. It can be viewed as if the processing instructions are already contained within the XML document. Since the context information is already available within the document, tools like search engines and data mining tools can be enhanced to use the information and yield more efficient results.

Syntax	No. of bits	Mnemonic

HelloBits {
  Bits
}
	8	uimsbf

(a)

```
class HelloBits {
  unsigned int Bits;
  void get() {
    Bits = ::getuint(8);
  }
};
```

(b)

```
class HelloBits {
  unsigned int(8) Bits;
};
```

(c)

Figure 1: `HelloBits`. (a) Representation using the MPEG-1/2 methodology. (b) Representation using C++ (A similar construct would be used for Java as well). (c) Representation using Flavor.

XFlavor is an extended version of the Flavor translator that provides XML features. In addition to the bitstream parsing and generating code (in C++ or Java), XFlavor also generates XML schema as well as the code for generating XML documents. With the XML document generating code, a bitstream representing multimedia data can be read in and a corresponding XML document can be automatically generated for easier and more flexible manipulation of the data. Also, in order to make the XML representation of the multimedia data strictly conforming to the original bitstream specification, the corresponding XML schema can be used. The schema [6] defines the syntax of the data and this is used as the guideline for determining the "validity" of XML represented data. Also, with the schema generated from the bitstream specification (Flavor description), the conforming XML documents can be exactly converted back into the original bitstream format. The converted data, then, can be processed in the bitstream domain (which yields in faster processing) by the bitstream specification compliant decoders.

In this paper, we first give a brief introduction of Flavor in Section 2. Then, in Section 3, XFlavor is described, and in Section 4, a list of possible applications using the XML representation is given. The benefits of using the representation are explained as well. Finally, we conclude with Section 5, summarizing the features and possible future extensions of XFlavor. More detailed information about Flavor and XFlavor can be found at **`http://flavor.sourceforge.net`**. The source code for the software is available as an open source project and the complete Flavor package (including XFlavor) can be downloaded from **`http://www.sourceforge.net/projects/flavor`**.

2. FLAVOR

Flavor is a new language developed for describing and processing bitstreams according to a specified syntax, and it's an acronym for Formal Language for Audio-Visual Object Representation [1, 2, 3]. As the name suggests, Flavor is designed as the framework for completely describing the compressed bitstream-level representation of audiovisual objects and giving an object-oriented view of the series of bits. Using Flavor, we have successfully described a variety of formats including MPEG-2 [7] bitstreams (Audio, Video and Systems). Also note that Flavor is used in the Structured Audio and Systems parts of the well-known MPEG-4 [8] standard.

2.1 Overview

Flavor extends C++ and Java in which its typing system also incorporates bitstream syntax. Thus, Flavor is designed as an object-oriented language with the concepts that are pertinent for bitstream-based media representation introduced. Additionally, Flavor has been explicitly designed to follow a declarative approach to bitstream syntax specification. This means that the designer is specifying how the data is laid out in the bitstream, and does not detail a step-by-step procedure that parses it.

The following trivial example - `HelloBits` in Figure 1 - shows how the integration of type and bitstream representation is accomplished using Flavor. `HelloBits` is a simple object with just a single value called `Bits`, represented using 8 bits. Using the MPEG-1/2 methodology, this would be described as shown in Figure 1(a). A C++ description of this single-value object would include a method to read its value and, have a form similar to the one shown in Figure 1(b). Here `getuint()` is assumed to be a function that reads bits from the bitstream (here 8) and returns them as an unsigned integer (the most significant bit first). When `HelloBits::get()` is called, the bitstream is read in and the resultant quantity is placed in the data member `Bits`. The same description in Flavor is shown in Figure 1(c).

As we can see, in Flavor the bitstream representation is integrated with the type declaration. The Flavor description should be read as: `Bits` is an unsigned integer quantity represented using 8 bits in the bitstream. The description indicates that when the system has to parse a `HelloBits` data type, it will just read the next 8 bits as an unsigned integer and assign them to the variable `Bits`.

The example discussed above, although trivial, demonstrates the differences between the various approaches. In Figure 1(a), we just have a tabulation of the various bitstream entities, grouped into syntactic units. This style is sufficient for straightforward representations, but fails when more complex structures are used (e.g., variable-length codes). In Figure 1(b), the syntax is incorporated into hand-written code embedded in a `get()` or an equivalent method. As a result, the syntax becomes an integral part of the decoding method even though the same decoding mechanism could be applied to a large variety of similar syntatic constructs. Also, it quickly becomes overly verbose.

Flavor comes with a wide range of facilities to define sophisticated bitstreams, including `if-else`, `switch`, `for`, and `while` constructs. In constrast with regular C++ or Java, these are all included in the data declaration part of the class, so they are completely disassociated from code that belongs to class methods. This is in line with the declarative nature of Flavor, where the focus is on defining the structure of the data, not operations on them. Additionally, Flavor comes with a translator that generates C++ or Java

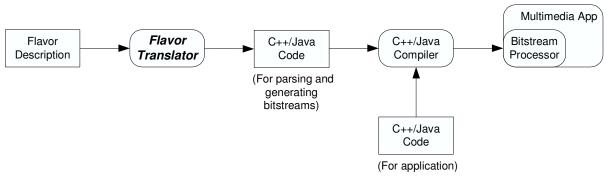

Figure 2: Using the Flavor description and translator to create the bitstream processing part of a multimedia application.

code from Flavor descriptions and the code includes methods (`get()` and `put()`) that can read or write data that complies to the Flavor-described representation. Figure 2 shows the steps taken to create a multimedia application using Flavor.

2.2 Features of Flavor

There are a number of features that the current version of Flavor provides and they are briefly described in this subsection:

A. Flavor can be used as a media representation documenting tool, substituting ad-hoc ways of describing a bitstream's syntax with a well-defined and concise language. Past approaches for syntax description utilized a combination of tabular data, pseudo-code and textual description to describe the format at hand. For example, MPEG-1 and 2 specifications are described using C-like pseudo-code syntax coupled with explanatory text and tabular data [9, 7]. Using such text can lead to ambiguities, and thus must be carefully crafted and tested over time. Flavor comes with unambiguous definition of all syntactic structures of the bitstream, eliminating the need for explanatory text.

B. The Flavor translator can be used to automatically generate C++ and Java code from the Flavor description. Such generated code can readily be used as bitstream generators, verifiers, parsers as well as a substantial portion of an encoder or decoder. In a standard or manual, if the bitstream syntax is described using Flavor, software developers can just cut and paste the Flavor description part into a text file, and then just feeding the file into the translator can generate a working code. This greatly simplifies and speeds up the development of software that processes coded audio-visual or general multimedia information.

Note that general-purpose programming languages such as C++ and Java do not provide native facilities for coping with bitstream-oriented data. Software codec or application developers need to build their own facilities, involving three components. First, they need to develop software that deals with the bitstream-oriented nature of data, as general-purpose microprocessors are strictly byte-oriented. Second, they need to implement parsing and generation code that complies with the syntax of the format at hand (be it proprietary or standard). Third, the parsing and generation code must account for different machine architectures so that consistent representation of bitstreams is guaranteed. These three tasks represent a significant amount of the overall development effort. They also have to be duplicated by everyone who requires access to a particular compressed representation within their application. Furthermore, they can also represent a substantial percentage of the overall execution time of the application [1, 2].

C. Flavor also allows for description of entropy-encoded data (e.g. Huffman code) and then produces optimized code for encoding/decoding the codewords. This feature is a must in media representation because most of the multimedia information - GIF, JPEG and MPEG - is encoded in a highly efficient form via source coding operation for minimizing the cost of storage and transmission [10]. In Flavor, the table of variable-length codes (VLCs) with corresponding values can be just put into a similar structure called *map* and an optimal code for efficiently processing/producing VLCs is created. Also, the translator checks whether the specified VLC table is uniquely decodable and the users can be assured of extremely fast decoding with minimal memory requirement due to the optimized code generation [3]. In addition, the development effort and time in creating software to process VLCs is greatly saved.

D. Flavor provides run-time API with a well-defined set of methods for bitstream I/O. Flavor already provides a run-time library to support file-based I/O for both C++ and Java. It is important to note that Flavor makes minimal assumption about the operating environment for the generated code. With the big and little-endian architectures coexisting in current computing machines, portability becomes an issue. Flavor's run-time API provides the necessary abstraction layer so that machine architecture no longer becomes a concern. Also, for designing a user-defined, application-specific I/O structures, Flavor provides an abstract class and new I/O structures just need to be inherited from the class.

E. Flavor allows immediate access to the content by any application developer, for such diverse use as editing, searching, indexing, filtering, etc. Having the capability to directly access the compressed multimedia information (bitstream) in a seamless fashion is very important, as more and more people are getting involved in accessing, creating, manipulating and distributing content.

F. Flavor can be used to redefine the syntax of content in both forward and backward compatible ways. The separation of parsing from the remaining coding/decoding operations allows its complete substitution as long as the interface (the semantics of the previously defined fields) remains the same. Old decoding code will simply ignore the new variables, while newly written encoders and decoders will be able to use them. This can eliminate the rigidity of current standards, where even a slight modification of the syntax to accommodate new techniques or functionalities render the content useless in non-flexible but nevertheless compliant decoders.

3. XFLAVOR

XFlavor is an extended version of Flavor that supports XML. It's a software tool extended from the Flavor translator in order to provide the following two additional functions: 1. Given a Flavor description of bitstream syntax, a corresponding XML schema can be created. 2. Given a Flavor description of bitstream syntax, XML document generating code can be created. Also, as a part of XFlavor, a software tool (*Bitstream Generator*) for converting an XML representation of multimedia data back into the bitstream representation is created.

Figure 3 illustrates the functions and some of the possible applications of XFlavor. Starting with a Flavor description of a multimedia bitstream (*Flavor File A*), XFlavor can generate an XML schema (*XSM Schema A*) or XML document generating code (*XML Doc Generator*). Then, applications can use the generated schema to create new multimedia content in the specified XML form (e.g., *Multimedia App A*). Additionally, using the XML document generating code, the bitstream representation of multimedia content can be converted into an XML document for more flexible and extensible processing of the data. Once the data is in the XML form, different applications can easily manipulate it using XML processors (e.g., *Multimedia App C and D*) or XML tools can be used to manipulate the data (e.g. XSLT processor). Also, for space and time critical applications, the XML representation of the data can be converted into bitstream representation (e.g., *Multimedia App B*).

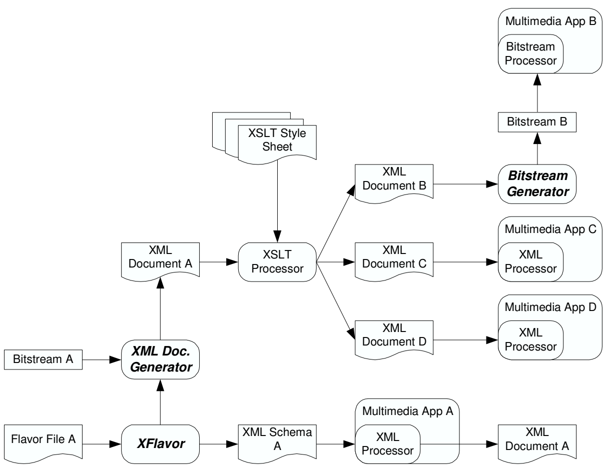

Figure 3: The functions of XFlavor used for different applications.

The main goal of XFlavor is to offer an alternate form of the multimedia data with the goal of providing the necessary structure and information so that different and interoperable applications can be created for the same data even if the expected syntax of the data is different among the applications. XML has been proven to be the solution for providing such goal in the IT world and XFlavor, by providing an XML representation of the multimedia data, enables the world of media representation to take advantage of such a solution.

XML adds an abstract layer by hiding the bitstream representation with object representation, aiding software developers and programmers in developing applications. With a vast number of XML processors/parsers available, and the XML document already providing semantic values of the bits in the bitstream, the application developers only need to focus their effort on semantic decoders. XFlavor, like Flavor, is based on *principal of separation* between data parsing operations and encoding, decoding and other operations. In the following subsections, we describe in detail the functions of XFlavor as well as their development issues.

3.1 Creating XML Schema

XML schema [6] is used to define the syntax of XML documents by specifying how the elements are laid out in the document as well as the attributes that are allowed with the elements. It also sets the data types for the content of the elements. The information in the schema is usually used to check the validity of the XML documents. The schema can be thought of as a set of rules that applications must agree on in creating/manipulating data so that their output data can be interchanged among them.

Schema also works as a guide in creating content. As the structure of the data and their types are defined succinctly by the schema, a corresponding XML document can be easily created. Since each element that can appear in the document is already specified in the schema with proper ordering, an appropriate user interface can be easily derived from the schema for creating new multimedia data. As the schema is created from a bitstream specification, the XML document created based on the schema can be transformed into a compliant bitstream form. In essence, the schema can be used to easily create a content generation (or even authoring) tool.

Another advantage of using the schema is that the data checking part can be automated using a schema validator. The data checking process is a very important and tedious part of multimedia applications and using the schema saves much effort needed by the developers in coding the software. The only part that needs to be worked on is the actual decoding part of the application.

The rest of this subsection describes the methods used to convert Flavor description into XML schema. In the following discussions, we assume the knowledge of the Flavor language. Detailed information about the language can be found in [1, 2] or at the project web page: `http://flavor.sourceforge.net`.

3.1.1 Representing Classes

In Flavor, the actual bitstream of multimedia content is represented using *parsable variables*. These variables are defined as certain types (e.g., unsigned integer in big-endian byte-ordering) along with the number of bits used in the bitstream. For example, in Figure 4(a), a parsable variable `length` of type unsigned integer in little-endian byte-ordering is defined and in the bitstream it is represented with 24 bits. This type of variable, which represents a specific field in the bitstream syntax, can be directly represented in an XML document using an element as defined in the schema of Figure 4(b). Note that a set of attributes is defined to identify the type of the content of the element. Defining the element as "`<xsd:element name="length" type="xsd:unsignedInt">`" is not sufficient because through the XML parser, the content is available only as a string and the type information is needed to convert the string into corresponding semantic value. Hence, the four attributes are used as a means to describe the data type. We call the four attributes *data type attributes* and the elements that are associated with the attributes are called *parsable elements*.

The `type` attribute is used to indicate one of the four data types available in Flavor: `char`, `int`, `float` and `double`. Flavor also supports the special `bit` type, however, this is converted into the `int` type. The `big` attribute is used for specifying the byte-ordering and the `sign` attribute is used for specifying whether or not the first bit is used as a sign bit. Lastly, the `bitLen` attribute specifies the number of bits used to represent the content in the bitstream.

```
little unsigned int(24) length;
```

(a)

```
<xsd:element name="length">
  <xsd:complexType>
    <xsd:simpleContent>
      <xsd:extension base="xsd:unsignedInt">
        <xsd:attribute name="type" type="xsd:unsignedInt" fixed="2"/>
        <xsd:attribute name="big" type="xsd:boolean" fixed="false"/>
        <xsd:attribute name="sign" type="xsd:boolean" fixed="false"/>
        <xsd:attribute name="bitLen" type="xsd:unsignedInt" fixed="24"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
</xsd:element>
```

(b)

Figure 4: (a) A declaration of a parsable variable. (b) An XML schema corresponding to the parsable variable defined in (a).

Since Flavor is designed as an object-oriented language, all the parsable variables are declared within a class. Thus, we need a way to describe Flavor classes and at the same time support all the object-oriented features related to them. First, the classes can be described using *complex types* in XML schema. Within the type, a number of parsable elements can be described in the order they will appear in the corresponding documents. This is similar to the Flavor class where a number of parsable variables can be defined in the order they'll appear in the bitstream. Second, the schema allows derivation of types (simple or complex) using the constructs: `xsd:extension` or `xsd:restriction`. These can be used to mimic inheritance as well as polymorphism as described below.

In Figure 5, a base class for all program descriptors in the MPEG-2 Systems specification [11] along with the Registration and Data Alignment descriptors, which are inherited from the base class, are defined in Flavor. By declaring the class variable, `descriptor`, a bitstream parsing (generating) code is generated, which depending on the *object identifier* value (`tag`), a corresponding class object is called. For example if the first 8 bits read in from the bitstream is equal to 5 (`tag` = 5), then, the rest of the bitstream is parsed according to the syntax described by the `RegistrationDescriptor` class. Else if `tag` = 6, then the `DataAlignmentDescriptor` class is used to parse the rest of the bits.

Figure 6 is the XML schema description of the same bitstream syntax described in Figure 5. In the schema, the `xsd:extension` construct is used to derive other types from the base complex type. The element `descriptor` is defined as the type `BaseProgramDescriptor`, and in an XML document, this element can take any one of the three types defined in the figure. Thus, as in Flavor or any other object-oriented language, inheritance and polymorphism are sup-

```
class BaseProgramDescriptor : unsigned int(8) tag = 0
{
  unsigned int(8) length;
}

class RegistrationDescriptor extends BaseProgramDescriptor : unsigned int(8) tag = 5
{
  unsigned int(32) format_identifier;
  int bytes_read;
  for (bytes_read=4; bytes_read < length; ) {
    bit(8) additional_identification_info;
    bytes_read += 1;
  }
}

class DataAlignmentDescriptor extends BaseProgramDescriptor : unsigned int(8) tag = 6
{
  unsigned int(8) alignment_type;
}

BaseProgramDescriptor descriptor;
```

Figure 5: A Flavor description of a base class (*BaseProgramDescriptor*) and two classes (*RegistrationDescriptor* and *DataAlignmentDescriptor*) that are derived from it. The object corresponding to any one of the three classes can be declared using the base class as the type.

ported.

In the Flavor description, the object identifiers (IDs) are used to distinguish between polymorphic parsable classes and they are read in from (or written into) the bitstream. Thus, they are also parsable variables and in the schema, like any other parsable variables, the IDs are represented as parsable elements. Also, the `for` loop is replaced with the construct - `<xsd:sequence minOccurs="0" maxOccurs="unbounded">` - because the loop can occur at least 0 times but the upper bound is not yet known. In the current version of XFlavor, the boolean constraint of the `for` loop is not checked because the current version of XML schema does not support such constraints. However, other existing tools can be used to extend it. For example, XSLT [12] can be used in complement to the schema so that tighter constraints are described and validated. Similarly, the `if` and `case` statements are represented with `<xsd:sequence minOccurs="0">` (by default `maxOccurs="1"`), the `while` statement with `<xsd:sequence minOccurs="0" maxOccurs="unbounded">`, and finally, the `do` ... `while` statement is described with `<xsd:sequence maxOccurs="unbounded">`.

Note that non-parsable variables are not part of the bitstream, thus, ignored from the schema.

3.1.2 Representing Maps

In addition to parsable variables, the bitstream can also be represented using a construct called map. Map represents VLCs and it's just a simple table with codewords and corresponding values. With the `BPC` map in Figure 7(a), if the first bit in the bitstream is 0, then a `YUVblocks` object, with the values {14, -1, 1} for its member variables, is assumed.

In a map, every codeword has to be unique and the uniqueness is determined by both by its bit length and integer value. For example, the codeword 0b110 can be represented as an integer value 6 with the bit length of 3. Since every map declaration (e.g. `YUVblocks(BPC) chroma_format;`) cor-

```
<xsd:complexType name="BaseProgramDescriptor">
  <xsd:sequence>
    <xsd:element name="tag">
      <xsd:complexType>
        <xsd:simpleContent>
          <xsd:extension base="xsd:unsignedInt">
            <xsd:attribute name="type" type="xsd:unsignedInt" fixed="2"/>
            <xsd:attribute name="big" type="xsd:boolean" fixed="true"/>
            <xsd:attribute name="sign" type="xsd:boolean" fixed="false"/>
            <xsd:attribute name="bitLen" type="xsd:unsignedInt" fixed="8"/>
          </xsd:extension>
        </xsd:simpleContent>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="length">
      <xsd:complexType>
        <xsd:simpleContent>
          <xsd:extension base="xsd:unsignedInt">
            <xsd:attribute name="type" type="xsd:unsignedInt" fixed="2"/>
            <xsd:attribute name="big" type="xsd:boolean" fixed="true"/>
            <xsd:attribute name="sign" type="xsd:boolean" fixed="false"/>
            <xsd:attribute name="bitLen" type="xsd:unsignedInt" fixed="8"/>
          </xsd:extension>
        </xsd:simpleContent>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="RegistrationDescriptor">
  <xsd:complexContent>
    <xsd:extension base="BaseProgramDescriptor">
      <xsd:sequence>
        <xsd:element name="format_identifier">
          <xsd:complexType>
            <xsd:simpleContent>
              <xsd:extension base="xsd:unsignedInt">
                <xsd:attribute name="type" type="xsd:unsignedInt" fixed="2"/>
                <xsd:attribute name="big" type="xsd:boolean" fixed="true"/>
                <xsd:attribute name="sign" type="xsd:boolean" fixed="false"/>
                <xsd:attribute name="bitLen" type="xsd:unsignedInt" fixed="32"/>
              </xsd:extension>
            </xsd:simpleContent>
          </xsd:complexType>
        </xsd:element>
        <xsd:sequence minOccurs="0" maxOccurs="unbounded">
          <xsd:element name="additional_identification_info">
            <xsd:complexType>
              <xsd:simpleContent>
                <xsd:extension base="xsd:unsignedInt">
                  <xsd:attribute name="type" type="xsd:unsignedInt" fixed="2"/>
                  <xsd:attribute name="big" type="xsd:boolean" fixed="true"/>
                  <xsd:attribute name="sign" type="xsd:boolean" fixed="false"/>
                  <xsd:attribute name="bitLen" type="xsd:unsignedInt" fixed="8"/>
                </xsd:extension>
              </xsd:simpleContent>
            </xsd:complexType>
          </xsd:element>
        </xsd:sequence>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="DataAlignmentDescriptor">
  <xsd:complexContent>
    <xsd:extension base="BaseProgramDescriptor">
      <xsd:sequence>
        <xsd:element name="alignment_type">
          <xsd:complexType>
            <xsd:simpleContent>
              <xsd:extension base="xsd:unsignedInt">
                <xsd:attribute name="type" type="xsd:unsignedInt" fixed="2"/>
                <xsd:attribute name="big" type="xsd:boolean" fixed="true"/>
                <xsd:attribute name="sign" type="xsd:boolean" fixed="false"/>
                <xsd:attribute name="bitLen" type="xsd:unsignedInt" fixed="8"/>
              </xsd:extension>
            </xsd:simpleContent>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:element name="descriptor" type="BaseProgramDescriptor"/>
```

Figure 6: An XML schema with type inheritance corresponding to the Flavor description with class inheritance in Figure 5.

```
map BPC(YUVblocks) {
  0b0,    {14, -1, 1},
  0b10,   {-4, 22, 2},
  0b110, {4, 2, -33}
}

map A(float) {
  0b1,    1.3,
  0b01,   2.2,
  0b001,  3.3,
  0b0001, float(32)
}
```

(a)

```
<xsd:complexType name="BPC">
  <xsd:sequence>
    <xsd:element name="code">
      <xsd:complexType>
        <xsd:simpleContent>
          <xsd:extension base="xsd:unsignedInt">
            <xsd:attribute name="bitLen" type="xsd:unsignedInt"/>
          </xsd:extension>
        </xsd:simpleContent>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
```

(b)

```
<xsd:complexType name="A">
  <xsd:sequence>
    <xsd:element name="code">
      <xsd:complexType>
        <xsd:simpleContent>
          <xsd:extension base="xsd:unsignedInt">
            <xsd:attribute name="bitLen" type="xsd:unsignedInt"/>
          </xsd:extension>
        </xsd:simpleContent>
      </xsd:complexType>
    </xsd:element>
    <xsd:sequence minOccurs="0">
      <xsd:element name="escape">
        <xsd:complexType>
          <xsd:simpleContent>
            <xsd:extension base="xsd:float">
              <xsd:attribute name="type" type="xsd:unsignedInt" fixed="3"/>
              <xsd:attribute name="big" type="xsd:boolean" fixed="true"/>
              <xsd:attribute name="sign" type="xsd:boolean" fixed="true"/>
              <xsd:attribute name="bitLen" type="xsd:unsignedInt" fixed="32"/>
            </xsd:extension>
          </xsd:simpleContent>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:sequence>
</xsd:complexType>
```

(c)

Figure 7: (a) Maps. *BPC* is a map that returns a class object - *YUVblocks* - depending on the codeword detected and *A* is a map that has an escape code defined. (b) An XML schema corresponding to the map *BPC* described in (a). (c) An XML schema corresponding to the map *A* defined in (a).

responds to a codeword in the bitstream, we just need to define, in the schema, a corresponding complex type with an element named `code` as shown in Figure 7(b). The `code` element contains an integer content that represents the integer value of the codeword and an attribute `bitLen` for indicating the bit length of the codeword. Again, tighter constraint can be defined using tools like XSLT so that only listed codewords are valid as the content of the `code` element. XSLT can also be used so that, given a codeword, the corresponding value is returned. That is, given the codeword 0b001 for the map `A` in Figure 7(a), a float value of 3.3 can be returned using XSLT.

Several alternatives to XSLT were also considered for defining all possible codewords of a given map. One alterna-

```
<xsd:choice>
  <xsd:element name="code" fixed="1">
    <xsd:complexType>
      <xsd:simpleContent>
        <xsd:extension base="xsd:unsignedInt">
          <xsd:attribute name="bitLen" type="xsd:unsignedInt" fixed="1"/>
        </xsd:extension>
      </xsd:simpleContent>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="code" fixed="1">
    <xsd:complexType>
      <xsd:simpleContent>
        <xsd:extension base="xsd:unsignedInt">
          <xsd:attribute name="bitLen" type="xsd:unsignedInt" fixed="2"/>
        </xsd:extension>
      </xsd:simpleContent>
    </xsd:complexType>
  </xsd:element>
</xsd:choice>
```

Figure 8: An invalid schema for representing a map. elements with the same name (*code*) and different types are not allowed in the same scope.

tive was just listing out the codewords in the schema as in Figure 8. However, in XML schema, elements cannot have the same name in the same scope if their types are not the same. Thus, the schema is rendered invalid by a schema validator because the `code` element is used more than once. Even though the type of the `code` elements in the figure seems to be the same, because the types are being defined within the element definition, the parser assumes that they would be of different type and the schema is rendered invalid. Alternatively, the `xsd:union` construct was considered so that each one of the possible codewords would be defined as a different type and all the types can be combined as the type for the `code` element. This way, we can defined just one `code` element and it would take any one of the defined types for the possible codewords. However, the types needed to be of complex type because two information are needed (bit length and integer value) and the `xsd:union` construct is valid only for simple types. Finally, the `xsd:list` construct was also considered where a list of two elements specifying the bit length and the integer value of the codeword can be defined. Within the `xsd:list` construct, the `xsd:enumeration` facet can be used to list the possible codewords. But, the facet does not allow setting a list of a pair of elements. Thus, the only way to list the possible codewords is using tools like XSLT.

As variable-length codeword lengths tend to get very large when their number increases, it is typical to specify *escape codes*, signifying that the actual value will be subsequently represented using a fixed-length code. In the map `A` described in Figure 7(a), when 4 bits with the integer value 1 is encountered in the bitstream, the actual return value for the map will be obtained by parsing 32 more bits. To accommodate such escape code, the schema can define a complex type with an extra element - `escape` - that is optional as shown in Figure 7(c).

3.2 XML Document Generation

Generating an XML document out of a Flavor described bitstream is straightforward. Every parsable variable read in from the bitstream corresponds to a parsable element in the XML document. For example, with the Flavor description displayed in Figure 9(a), we can be assured of the `length` field getting updated with a correct integer value as soon as the corresponding 24 bits are read in from the bitstream. Say the value read in was 128, then, the corresponding XML element named `length` can be generated as shown in Figure 9(b). The element does not have any type information because the XML document is generated assuming the existence of a corresponding schema. Alternatively, there's an option for generating all the necessary attributes along with each parsable element. With this option, the parsable element in Figure 9(b) would be represented as the one in Figure 9(c). This might be useful for the applications that do not follow the schema, and thus, the schema is not used.

```
int(24) length;
```
(a)

```
<length>128</length>
```
(b)

```
<length type="2" big="true" sign="true" bitLen="24">128</length>
```
(c)

Figure 9: (a) A parsable variable named *length*. (b) A parsable XML element corresponding to the *length* parsable variable described in (a). (c) The same XML element as the one in (b) but with the data type attribute values explicitly stated.

```
class HelloBits {
  int(24) length;
}

HelloBits hello;
```
(a)

```
<hello>
  <length>128</length>
</hello>
```
(b)

Figure 10: (a) A Flavor description of a simple class. (b) An XML representation of an instance of the Flavor description in (a).

With XFlavor, in addition to the `get()` and `put()` methods for reading and writing bitstreams, `putxml()` method is also created so that the bitstream can be parsed and a corresponding XML document can be generated. The method has a boolean input parameter that indicates whether or not we assume the existence of an XML schema for the document. Depending on this option, the `putxml()` method will generate XML elements as shown in Figure 9(b) or (c).

In Flavor, parsable variables are always defined within a class (a parsable class) and XFlavor makes sure that such relationship between the class and its member variables are maintained in the XML representation. With XFlavor, a parsable class is represented as an XML element (class element) and all the parsable variables within the class are represented as parsable elements within the class element. Thus, a compound element is created for all the parsable classes defined in the Flavor description. For example, Figure 10(a) and (b) show a simple class definition in Flavor and an instance of the class represented in XML, respectively.

Additionally, parsable variables follow a special scoping rule; within a class, all parsable variables are considered as class member variables, regardless of the scope they are encountered in. Therefore, all the parsable variables are defined as an element right under the class element. As an example, a Flavor description of an MPEG-2 Program Stream System header [11] and the XML document representing an instance of the Flavor description, as generated by the `putxml()` method, are displayed in Figures 11(a) and (b). Figure 11(b) is an XML document generated assuming the existence of the corresponding XML schema. Therefore, the data type attributes are not specified.

```
class SystemHeader
{
   StartCode          sh_start_code(SC_PS_Sys_Hdr);
   unsigned int(16)   header_length;
   bit(1)             marker = 0b1;
   unsigned int(22)   rate_bound;
   bit(1)             marker = 0b1;
   unsigned int(6)    audio_bound;
   bit(1)             fixed_flag;
   bit(1)             CSPS_flag;
   bit(1)             system_audio_lock_flag;
   bit(1)             system_video_lock_flag;
   bit(1)             marker = 0b1;
   unsigned int(5)    video_bound;
   bit(1)             packet_rate_restriction_flag;
   const bit(7)       reserved = 0x7F;
   bit(1)*            next_bit;
   while (next_bit == 0b1) {
      unsigned int(8)         stream_id;
      bit(2)                  bit_pattern = 0b11;
      bit(1)                  P_STD_buffer_bound_scale;
      unsigned int(13)        P_STD_buffer_size_bound;
      bit(1)*                 next_bit;
   }
}
```

(a)

```
<system_header>
   <sh_start_code>
      <sc_prefix>
         <sc_prefix>1</sc_prefix>
      </sc_prefix>
      <start_code>187</start_code>
   </sh_start_code>
   <header_length>12</header_length>
   <marker>1</marker>
   <rate_bound>7653</rate_bound>
   <marker>1</marker>
   <audio_bound>1</audio_bound>
   <fixed_flag>1</fixed_flag>
   <CSPS_flag>0</CSPS_flag>
   <system_audio_lock_flag>1</system_audio_lock_flag>
   <system_video_lock_flag>1</system_video_lock_flag>
   <marker>1</marker>
   <video_bound>1</video_bound>
   <packet_rate_restriction_flag>1</packet_rate_restriction_flag>
   <reserved>127</reserved>
   <stream_id>224</stream_id>
   <bit_pattern>3</bit_pattern>
   <P_STD_buffer_bound_scale>1</P_STD_buffer_bound_scale>
   <P_STD_buffer_size_bound>80</P_STD_buffer_size_bound>
   <stream_id>192</stream_id>
   <bit_pattern>3</bit_pattern>
   <P_STD_buffer_bound_scale>0</P_STD_buffer_bound_scale>
   <P_STD_buffer_size_bound>80</P_STD_buffer_size_bound>
</system_header>
```

(b)

Figure 11: (a) The Flavor representation of the MPEG-2 Program Stream System header. (b) The XML representation of an instance of the MPEG-2 Program Stream System header.

3.3 Bitstream Generation

As a part of XFlavor, we have developed a software tool called *Bitstream Generator* that converts an XML representation of multimedia data into a bitstream representation. The current software only works with XML documents that have the data type attributes of the parsable elements explicitly stated. It's a simple parsing tool where, based on the data type attributes, the content of each parsable element is converted into a bitstream representation. For example, the content of the parsable element - `length` - in Figure 9(c) would yield 0x000080 in hex. First, the content 128 is read in as a string and then using the values of the `type` and `sign` attributes, it's converted into a signed integer. Then, using the `big` and `bitLen` attributes, the signed integer is converted into a bitstream representation of 24 bits with the most significant byte first. For the future extension, we are working with an XML parser so that, as long as the schema is available, we can get the data type attributes and the parsable elements can be of the compact form as shown in Figure 9(b).

4. APPLICATIONS AND BENEFITS

In the following subsections, we describe some of the possible applications of XFlavor as well as the benefits obtained by using the XML features.

4.1 Content Extraction

XML representation allows easy access to different elements. It allows an application to modify and add/delete elements of an XML document, and yet, allows different applications to still work with the modified document. In other words, even if the structure of the document is changed, the correct semantics of the original elements can still be obtained. This means, any application can add proprietary information into the document without ever worrying about breaking the semantics of the original data, thus preserving the interoperability of the data. This is the beauty of a self-describing format.

An XML representation allows keeping all the information available in the encoder side and the applications decide how much of the built-in content intelligence it wishes to use. In the field of content extraction, many software tools have been developed to extract meaningful information from the content. For example, the score information from the caption of a sports video can be obtained using various signal processing tools. However, such information is already present in the encoder side and it can be preserved in the XML document. Any application requiring such information will be able to use it and other applications can simply ignore it.

Additionally, XML based applications offer benefits such as providing support for different bitstream syntax without reprogramming/recompiling. An application built for Syntax A can still work with a document with Syntax B as long as the required elements are still available in the document and the tags remain the same.

4.2 Transcoder

With XML representation, all the XML related technologies and software tools can be directly used. One very important XML technology is XSLT. XSLT is a language defined using XML for transforming XML documents. With XML alone, interoperability is only obtained among the applications that understand a certain set of predefined tags. However, with XSLT, any application can be enabled to use any XML document as long as the actual data in the document is useful to the application. This "universal interoperability" is achieved by transforming XML documents into different structures usable for different applications.

Using such technology, a transcoder can be easily created by simply declaring a set of rules to transform the source elements. The style sheet (containing XSLT rules) along with the source document can be fed into a XSLT processor to generate a new document with desired structure and information. Also, a set of style sheets can be used to manipulate the elements of a given document.

4.3 Compression

XFlavor (Bitstream Generator) has the ability to transform a given XML representation of multimedia content back into the bitstream representation. With the very high usage of XML applications, it's very important to come up with an efficient representation/processing of XML documents. Because XML is text based it may not be efficient enough in the compression as well as processing sense. To fix this problem, binary representation of the content can be created for the given XML representation.

MPEG-7 [13] has already standardized binary encoding of its XML documents and for wireless applications where resources are very scarce, a binary version of XML is being deployed (WBXML [14]). Additionally, there are also a number of XML compression tools such as XMill [15] and XMLZip [16]. Though these can be applied to the XML representation of multimedia data, they do not yield as efficient result as the original compressed bitstream.

XMill is a compression tool based on a novel compression algorithm derived from the "grouping strategy" [15]. This algorithm sometimes yields better compression result than conventional compression (e.g., *gzip*) on the original data, however, this is usually the case for the text data. For multimedia data originally represented by compressed bitstream, it's very difficult to reduce the corresponding XML document to a size smaller than the original data. XMLZip is also a compression tool developed by XML Solutions (now a part of Vitria Technology) for compressing XML documents while maintaining the XML structure. As expected, it yields worse compression result than XMill, but it offers DOM API [17] in the compressed domain, which ultimately results in faster XML parsing and processing.

Table 1 compares the compression results of XMill and XMLZip with the original bitstream.[1] With XMLZip, the size of the XML document can approximately be reduced to twice the size of the original data. XMill, with the Compression Factor (CF) set to the highest value (CF = 9), returns a compressed XML document that is about 30% bigger than the original data size. If *bzip* is used instead of *gzip* in XMill and if the file size is big, the compressed document becomes less than 5% bigger than the original data.

[1]BiM [13, 18] and Millau [19] were also looked into, however, the encoders and decoders weren't available for testing at the time of writing this paper.

Table 1: Compression results for different XML compressors.

	File 1	**File 2**
Original File Size (in Bytes)	10,048	278,583
XML File Size (in Bytes)	426,262	11,264,591
XMLZip File Size (in Bytes)	21,898	496,957
XMill (gzip, CF = 6) (in Bytes)	14,539	364,455
XMill (gzip, CF = 9) (in Bytes)	14,492	359,611
XMill (bzip) (in Bytes)	11,223	285,269

With XFlavor, the original file size can be achieved, whereas, XMill and XMLZip both yield bigger file sizes. This is due to the fact that we are dealing with a compressed data to begin with and any additional compression on the original data would be very difficult. If storage or transmission is the main concern then the original bitstream should be used and upon the time of processing, the bitstream can be converted to an XML file. It's also beneficial to maintain the bitstream representation of the data allowing fast processing for the applications like *Multimedia App B* in Figure 3.

XMLZip, though producing worse compression result than XMill, offers the same benefit of using the XML format in the compressed domain by preserving the structure of the data. A special XML parser can be used to access and manipulate the compressed XML data. On the other hand, XMill only offers compression and for large XML data, the size of the compressed data can be very close to the original data size. However, for processing of the data, the compressed XML data has to be uncompressed to recover the XML structure. In this sense, XMLZip can be more useful than XMill if compression is not the main issue. But, with XFlavor, the XML data can be converted to the original bitstream syntax and the resulting data can be processed using existing decoders that conform to the given bitstream specification. Additionally, XFlavor converts the data in XML format into the original (compressed) bitstream and the size of this bitstream is always smaller than the same XML data compressed using XMill.

5. CONCLUSION

Traditionally, the objective of media representation has been compression where the cost of storage and transmission has been the major driving force. For this reason, multimedia information is encoded in a highly efficient binary form, removing all the redundant information. However, the storage unit and transmission bandwidth have gotten cheaper and with high capacity computers widely available, there is a high demand for applications that offer features other than just playback of media data.

To support applications with a number of functionalities as editing, indexing, annotating, etc., it is important to create an abstract layer to process, manipulate and create multimedia content without actually manipulating the bits. XML provides the necessary abstraction, which allows faster and easier development of applications for such media processing. The XML representation hides the bitstream structure from the applications, and the applications are able to work with the given data as long as all the necessary semantic values are present in the data. This yields in extremely flexible and extensible representation of data where different applications can add/delete extra information to the data and still maintain interoperability. In complement to this, for applications that require small data size and fast processing of the data, XFlavor can compress the XML representation of the data back into its native bitstream form. Thus, with XFlavor, the compression, flexibility and extensibility issues can be satisfied.

In [20], Jon Bosak states "XML can do for data what Java has done for programs, which is to make the data both platform-independent and vendor-independent." This is the reason why XML is becoming the standard for data exchange as well as media-independent publishing. XFlavor, by providing the XML features, applies the benefits provided by XML in the IT world into the world of media representation.

In summary, representing multimedia data in XML format causes the data to be self-describing. Each field of the data is represented as an XML element, which includes the context information of the field. In this case, adding or deleting extra element does not change the context information of the field, thus proper decoding of the data can still be maintained. This is not possible with the original bitstream representation of the data because the context information of a field is derived from the structure of the bitstream. Thus, adding or deleting extra information breaks the structure and changes the whole meaning of the field. Additionally, due to the self-describing nature of the XML documents, media data in XML format can be easily filtered in the data mining process. With the ever increasing number of available resources, efficient data mining tool is of a very importance.

As future extension, we plan to enhance XFlavor to generate XSLT stylesheets as a complement to the schema. The Bitstream Generator will also be enhanced to use XML parser so that even if the data type attributes are not explicitly stated in the document, as long as the appropriate schema is referenced, the correct bitstream can be produced. As another possible future extension of XFlavor, a function for producing a Flavor description given an XML schema or document can be created so that a corresponding bitstream parsing/generating code can be automatically generated. With this, any multimedia data can be created and modified as an XML document, and if needed, it can be converted to an equivalent bitstream format along with the corresponding bitstream parser/generator.

6. REFERENCES

[1] Y. Fang and A. Eleftheriadis. A syntactic framework for bitstream-level representation of audio-visual objects. In *3rd IEEE International Conference on Image Processing*, Proceedings, pages II.426–II.432, Lausanne, Switzerland, September 1996.

[2] A. Eleftheriadis. Flavor: A language for media representation. In *ACM International Conference on Multimedia*, Proceedings, pages 1–9, Seattle, WA, USA, November 1997.

[3] Y. Fang and A. Eleftheriadis. Automatic generation of entropy coding programs using Flavor. In *IEEE 2nd Workshop on Multimedia Signal Processing*, Proceedings, pages 341–346, Redondo Beach, CA, USA, December 1998.

[4] W3C Recommendation. *Extensible Markup Language (XML) 1.0 (Second Edition)*, October 2000.

[5] CompuServe Inc. *Graphics Interchange Format*, 1987 and 1989.

[6] W3C Recommendation. *XML Schema Part 0: Primer, XML Schema Part 1: Structures, XML Schema Part 2: Datatypes*, May 2001.

[7] ISO/IEC 13818 International Standard (MPEG-2). *Information technology – Generic coding of moving pictures and associated audio information*, 1996.

[8] ISO/IEC 14496 International Standard (MPEG-4). *Information technology – Coding of audio-visual objects*, 1999.

[9] ISO/IEC 11172 International Standard (MPEG-1). *Information technology – Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbits/s*, 1993.

[10] T. M. Cover and J. A. Thomas. *Elements of Information Theory.* Wiley, 1991.

[11] ISO/IEC 13818-1 International Standard. *Information technology – Generic coding of moving pictures and associated audio information: Systems*, 1996.

[12] W3C Recommendation. *XSL Transformations (XSLT) Version 1.0*, November 1999.

[13] A. Avaro and P. Salembier. Mpeg-7 systems: Overview. *IEEE Transactions on Circuits and Systems for Video Technology*, 11(6):760–764, June 2001.

[14] W3C Note. *WAP Binary XML Content Format*, June 1999.

[15] H. Liefke and D. Suciu. XMill: An efficient compressor for XML data. In *2000 ACM SIGMOD International Conference on Management of Data*, Proceedings, pages 153–164, 2000.

[16] http://www.xmls.com. *XMLZip*.

[17] W3C Recommendation. *Document Object Model (DOM) Level 2 Specification*, November 2000.

[18] http://www.expway.tv. *BiM*.

[19] M. Girardot and N. Sundaresan. Efficient Representation and Streaming of XML Content over the Internet Medium. In *IEEE International Conference on Multimedia and Expo (ICME)*, volume 1 of *Proceedings*, pages 67–70, New York, NY, USA, July 2000.

[20] J. Bosak. Media-independent publishing: Four myths about XML. *IEEE Computer*, 31(10):120–122, October 1998.