Flavor: A Language for Media Representation

Alexandros Eleftheriadis and Danny Hong
Department of Electrical Engineering*
Columbia University
New York, NY 10027, USA

February 26, 2004

1 Introduction

Flavor, which stands for Formal Language for Audio-Visual Object Representation, originated from the need to simplify and speed up the development of software that processes coded audio-visual or general multimedia information. This includes encoders and decoders as well as applications that manipulate such information. Examples include editing tools, synthetic content creation tools, multimedia indexing and search engines, etc. Such information is invariably encoded in a highly efficient form to minimize the cost of storage and transmission. This source coding [1] operation is almost always performed in a bitstream-oriented fashion: the data to be represented is converted to a sequence of binary values of arbitrary (and typically variable) lengths, according to a specified syntax. The syntax itself can have various degrees of sophistication. One of the simplest forms is the GIF87a format [2], consisting of essentially two headers and blocks of coded image data using the Lempel-Ziv-Welch (LZW) compression. Much more complex formats include JPEG [3], MPEG-1 [4], MPEG-2 [5] and MPEG-4 [6], among others.

General-purpose programming languages such as C++ and Java do not provide native facilities for coping with such data. Software codec (encoder/decoder) or application developers need to build their own facilities, involving two components. First, they need to develop software that deals with the bitstream-oriented nature of the 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. These two 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 represent a substantial percentage of the overall execution time of the application.

Flavor addresses these problems in an integrated way. First, it allows the "formal" description of the bitstream syntax. Formal here means that the description is based on a well-defined grammar, and as a result is amenable to software tool manipulation. In the past, such descriptions were using ad-hoc conventions involving tabular data or pseudo-code. A second and key aspect of Flavor's architecture is that this description has been designed in a style similar to C++ and Java, both heavily used object-oriented languages in multimedia applications development. The similar syntax makes it accessible to users already familiar with C++ and Java. Additionally, Flavor descriptions can easily be converted to standard C++ or Java code, which can be compiled and used for parsing, verifying and generating bitstreams.

*This material is based upon work supported in part by the National Science Foundation under Grants MIPS-9703163 and ACI-0313116.

Flavor was designed as an object-oriented language, anticipating an audio-visual world comprised of audio-visual objects, both synthetic and natural, and combining it with well-established paradigms for software design and implementation. Its facilities go beyond the mere duplication of C++ and Java features, and introduce several new concepts that are pertinent for bitstream-based media representation.

In order to validate the expressive power of the language, several existing bitstream formats have already been described in Flavor, including sophisticated structures such as MPEG-1 Systems, Video and Audio. A translator has also been developed for translating Flavor code to C++ or Java code. Note that Flavor is currently used in the Structured Audio as well as the Systems parts of the MPEG-4 standard.

Emerging multimedia representation techniques can directly use Flavor to represent the bitstream syntax in their specifications. This will allow immediate use of such specifications in new or existing applications, since the code to access/generate conforming data can be generated directly from the specification with zero cost. In addition, such code can be automatically optimized; this is particularly important for operations such as Huffman encoding/decoding, a very common tool in media representation.

In the following, we first present a brief background of the language in terms of its history and technical approach. We then describe each of its features, including declarations and constants, expressions and statements, classes, scoping rules, maps, and built-in operators. We also describe the translator and its simple run-time API. Finally, we conclude with an overview of the benefits of using the Flavor approach for media representation. More detailed information and publicly available software can be found in the Flavor web site at: `http://flavor.sourceforge.net`.

2 Background

2.1 A Brief History

Flavor has its origins in a Perl script (`mkvlc`) [7] that was developed in early 1994 in order to automate the (laborious) generation of C code declarations for variable-length code (VLC) tables of the MPEG-2 Video specification[1]. In November 1995, the ideas behind `mkvlc` took a more concrete shape in the form of a "syntactic description language," [8, 9] i.e., a formal way to describe not just VLCs, but the entire structure of a bitstream. Such a facility was proposed to the MPEG-4 standardization activity, which at that time had started to consider flexible, even programmable, audio-visual decoding systems. The language subsequently underwent a series of revisions obtaining input from several participants in the MPEG-4 standardization activity, and its specification is now fairly stable.

2.2 Technical Approach

Flavor provides a formal way to specify how data is laid out in a serialized bitstream. It is based on a *principle of separation* between bitstream parsing operations and encoding, decoding and other operations. This separation acknowledges the fact that the same syntax can be utilized by different tools, but also that the same tool can work unchanged with a different bitstream syntax. For example, the number of bits used for a specific field can change without modifying any part of the application program.

Past approaches for syntax description utilized a combination of tabular data, pseudo-code, and textual description to describe the format at hand. Taking MPEG as an example, both MPEG-1

[1]The `mkvlc` program is available at `http://www.ee.columbia.edu/~eleft/software/mkvlc.tar.gz`.

and MPEG-2 specifications were described using a C-like pseudo-code syntax (originally introduced in 1990 by Milton Anderson, Bellcore), coupled with explanatory text and tabular data. Several of the lower and most sophisticated layers (e.g., macroblock) could only be handled by explanatory text. The text had to be carefully crafted and tested over time for ambiguities. Other specifications (e.g., GIF and JPEG) use similar bitstream representation schemes, and hence share the same limitations.

Similar to Flavor, facilities that simplify and speed up development of software have been proven to be invaluable in almost every domain. For example, lex and yacc [10] help to write programs such as translators and compilers that transform/process structured textual input, whereas Flavor helps to write programs that process structured binary input. Note that most of the modern translators and compilers are built using lex and yacc or similar tools.

A wide variety of facilities that help to write code for "marshalling", a fundamental operation in distributed systems where consistent exchange of structured data is ensured, are also available. Examples in this category include Sun's ONC XDR (External Data Representation) [11, 12] and the `rpcgen` compiler [13] which automatically generates marshalling code, as well as CORBA IDL [14], among others. Another widely used facility in this communications and networks area is ASN.1 (Abstract Syntax Notation 1) [15, 16]. Similar to these facilities, Flavor ensures consistent specification and representation of multimedia data. Two main differences are that in the former case: 1) the programmer does not have control over the data representation (the binary representation for each data type is predefined), and 2) the intricacies of source coding operations are not addressed. The facilities hide bitstream representation from the developer, whereas in our case the binary encoding is the actual target of description.

It is interesting to note that all prior approaches to syntactic description were concerned only with the definition of message structures typically found in communication systems. These tend to have a much simpler structure compared with coded representations of audio-visual information (compare the UDP packet header with the baseline JPEG specification, for example).

A new language, Bitstream Syntax Description Language (BSDL) [17, 18], has been introduced in MPEG-21 [19] for describing the structure of a bitstream using XML Schema [20]. However, unlike Flavor, BSDL is developed to address only the high-level structure of the bitstream, and it becomes impossible to fully describe the syntax on a bit-per-bit basis. For example, BSDL does not have a facility to cope with source coding operations, whereas Flavor does. Also, BSDL descriptions tend to be overly verbose. Note that the latest Flavor translator supports XML features and it can be used to translate Flavor descriptions into corresponding XML schemas [21].

Flavor was designed to be an intuitive and natural extension of the typing system of object-oriented languages like C++ and Java. This means that the bitstream representation information is placed together with the data declarations in a single place. In C++ and Java, this place is where a class is defined.

Flavor has been explicitly designed to follow a declarative approach to bitstream syntax specification. In other words, the designer is specifying how the data is laid out on the bitstream, and does not detail a step-by-step procedure that parses it. This latter procedural approach would severely limit both the expressive power as well as the capability for automated processing and optimization, as it would eliminate the necessary level of abstraction. As a result of this declarative approach, Flavor does not have functions or methods.

A related example from traditional programming is the handling of floating point numbers. The programmer does not have to specify how such numbers are represented or how operations are performed; these tasks are automatically taken care of by the compiler in coordination with the underlying hardware or run-time emulation libraries.

An additional feature of combining type declaration and bitstream representation is that the underlying object hierarchy of the base programming language (C++ or Java) becomes quite naturally the object hierarchy for bitstream representation purposes as well. This is an important benefit for ease of application development, and it also allows Flavor to have a very rich typing system itself.

Syntax	No. of bits	Mnemonic
HelloBits{		
Bits	8	uimsbf
}		

(a)

```
class HelloBits {
  unsigned int Bits;
  void get() {
    Bits = ::getuint(8);
  }
  void put() {
    ::putuint(Bits, 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 similiar construct would be used for Java as well). (c) Representation using Flavor.

"**HelloBits**" – Traditionally, programming languages are introduced via a simple "Hello World!" program, which just prints out this simple message on the user's terminal. We will use a similar example with Flavor, but here we are concerned about bits, rather than text characters. Figure 1 shows a set of trivial examples indicating how the integration of type and bitstream representation information is accomplished. Consider a simple object called `HelloBits` with just a single value, 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 two methods to read and write 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); the `putuint()` function has similar functionality but for output purposes. When `HelloBits::get()` is called, the bitstream is read 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. Note that there is no implicit encoding rule as in ASN.1: the rule here is embedded in the type declaration and 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`.

These examples, although trivial, demonstrate 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 – VLCs). In Figure 1(b), the syntax is incorporated into hand-written code embedded in a `get()` and `put()` or an equivalent set of methods. As a result, the syntax becomes an integral part of the encoding/decoding method even though the same encoding/decoding mechanism could be applied to a large variety of similar syntactic constructs. Also, it quickly becomes overly verbose.

Flavor provides a wide range of facilities to define sophisticated bitstreams, including `if-else`, `switch`, `for`, and `while` constructs. In contrast with regular C++ and 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. As we show later on, a translator can automatically generate C++ and/or Java methods (`get()` and `put()`) that can read/write data that complies to the Flavor-described representation.

In the following we describe each of the language features in more detail, emphasizing the differences between C++ and Java. In order to ensure that Flavor semantics are in line with both C++ and Java, whenever there was a conflict a common denominator approach was used.

3 Language Overview

3.1 Declarations and Constants

3.1.1 Literals

All traditional C++ and Java literals are supported by Flavor. This includes integers, floating-point numbers and character constants (e.g., '`a`'). Strings are also supported by Flavor. They are converted to arrays with or without a trailing '`\0`' (null character). Additionally, Flavor defines a special binary number notation using the prefix `0b`. Numbers represented with such notation are called binary literals (or bit strings) and, in addition to the actual value, also convey their length. For example, `0b011` denotes the number 3 represented using 3 bits. For readability, a bit string can include periods after every four digits, e.g., `0b0010.01`. Hexadecimal or octal constants used in the context of a bit string also convey their length in addition to their value. Whenever the length is irrelevant, a binary literal is treated as a regular integer literal.

3.1.2 Comments

Both multi-line `(/**/)` and single-line `(//)` comments are allowed. The multi-line comment delimiters cannot be nested.

3.1.3 Names

Variable names follow the C++ and Java conventions (e.g., variable names cannot start with a number). The keywords that are used in C++ and Java are considered reserved in Flavor.

3.1.4 Types

Flavor supports the common subset of C++ and Java built-in or fundamental types. This includes `char`, `int`, `float` and `double` along with all appropriate modifiers (`short`, `long`, `signed` and `unsigned`). Additionally, Flavor defines a new type called `bit` and a set of new modifiers, `big` and `little`. The type `bit` is used to accommodate bit string variables and the new modifiers are used to indicate the endianess of bytes. For example, the `little` modifier is used to represent the numbers using little-endian byte ordering; by default, big-endian byte ordering is assumed. Note that endianess here refers to the bitstream representation, not the processor on which Flavor software may be running. The latter is irrelevant for the bitstream description.

Flavor also allows declaration of new types in the form of classes (refer to Section 3.3 for more information regarding classes); however, Flavor does not support pointers, references, casts, or C++ operators related to pointers. Structures or enumerations are not supported either, since they are not supported by Java.

3.1.5 Declarations

Regular variable declarations can be used in Flavor in the same way as in C++ and Java. As Flavor follows a declarative approach, constant variable declarations with specified values are allowed everywhere (there is no constructor to set the initial values). Following the C++ approach, the `const` keyword is used and the declaration such as "`const int var = 1;`" is valid anywhere (not just in global scope). The two major differences are the declaration of parsable variables and arrays.

```
(a) [aligned(length)] [modifiers] type(size) variable [=value];
(b) int(24) var;
(c) aligned(8) unsigned int(24) var;
(d) int(24)* var;
(e) int(24) var = 2;
(f) little unsigned int(32) var = 0xFFFFFF10 .. 0xFFFFFFFF;
```

Figure 2: Parsable variable. (a) Parsable variable declaration syntax. (b) Declaration of a parsable variable. (c) Declaration of a byte-aligned parsable variable. (d) Declaration of a variable for a look-ahead parsing. (e) Declaration of a parsable variable with an expected value. (f) Declaration of a parsable variable with an expected range of values.

Parsable variables: Parsable variables are the core of Flavor's design; it is the proper definition of these variables that defines the bitstream syntax. These variables are the only ones whose values are written to and read from the bitstream. Parsable variables must include a parse size specification immediately after their type declaration, as shown in Figure 2. In Figure 2(a), the `size` argument can be an integer constant, a non-constant variable of type compatible to `int`, or a map (Section 3.5) with the same type as the variable. This means that the parse size of a variable can be controlled by another variable. Figure 2(b) illustrates declaration of a parsable variable `var` with parse sizes 24.

In addition to the parse size specification, parsable variables can also have the `aligned` modifier (in addition to all the supported modifiers). This signifies that the variable begins at the next integer multiple boundary of the length argument – `length` – specified within the alignment expression. If this length is omitted, an alignment size of 8 is assumed (byte boundary). Note that the `aligned` modifier must precede all other modifiers. For example, Figure 2(c) shows a parsable variable `var` that must begin at a byte boundary in the bitstream. Thus, for parsing, any intermediate bits are ignored, while for output bitstream generation the bitstream is padded with zeroes.

As we will see later on, parsable variables cannot be assigned to. This ensures that the syntax is preserved regardless if we are performing an input or output operation. However, parsable variables *can be redeclared*, as long as their type remains the same, only the parse size is changed, and the original declaration was not as a `const`. This allows one to select the parse size depending on the context (see Section 3.2). On top of this, they obey special scoping rules as described in Section 3.4.

In general, the parse size expression must be a non-negative value. The special value 0 can be used when, depending on the bitstream context, a variable is not present in the bitstream but obtains a default value. In this case, no bits will be parsed or generated, however, the semantics of the declaration will be preserved. The variables of type `float`, `double`, and `long double` are only allowed to have a parse size equal to the fixed size that their standard representation requires (32 bits for `float` and 64 bits for `double` and `long double`).

Look-ahead parsing: In several instances, it is desirable to examine the immediately following bits in the bitstream without actually removing the bits from the input stream. To support this behavior, a '`*`' character can be placed after the parse size parentheses. Note that for bitstream output purposes, this has no effect. An example of a declaration of a variable for look-ahead parsing is given in Figure 2(d). Note that the value for the variable `var` will be obtained by reading 24 bits from the bitstream, but the '`*`' character indicates that the bits should be left in the bitstream.

Parsable variables with expected values: Very often, certain parsable variables in the syntax have to have specific values (markers, start codes, reserved bits, etc.). These are specified as initialization values for parsable variables. Figure 2(e) shows an example, where `var` is an integer represented with 24 bits, and must have the value 2. That is, on reading 24 bits from the bitstream, they must correspond to the value 2 or the parsing (or the look-ahead parsing) will fail. Similarly, on writing into the bitstream, the value 2 will be encoded using 24 bits. The `const` modifier may be prepended in the declaration, to indicate that the parsable variable will have this constant value and, as a result, cannot be redeclared. Additionally, a parsable variable may be allowed to have a range of values and this can be specified as shown in Figure 2(f).

As both parse size and initial value can be arbitrary expressions, we should note that the order of evaluation is parse expression first, followed by the initializing expression.

Arrays: Arrays have special behavior in Flavor, due to its declarative nature but also due to the desire for very dynamic type declarations. For example, we want to be able to declare an array with different array sizes depending on the context. In addition, we may need to load the elements of a parsable array one at a time. The array size, then, does not have to be a constant expression (as in C++ and Java), but it can be a variable as well. The example in Figure 3(a) is allowed in Flavor.

```
int(5) var;  int A[2] = {1, 2};  int var = 1;                               int(2) A[[3]] = 1;
int A[var];  int B[3] = 5;       int((var++)) A[(var++)] = (var++);   int(4) B[[2]][3];
    (a)             (b)                              (c)                              (d)
```

Figure 3: Array. (a) Declaration with dynamic size specification. (b) Declaration with initialization. (c) Declaration with dynamic array and parse size. (d) Declaration of partial arrays.

An interesting question is how to handle initialization of arrays, or parsable arrays with expected values. In addition to the usual brace-expression initialization, Flavor also provides a mechanism that involves the specification of a single expression for the initializion as shown in Figure 3(b). This means that all elements of the array `B` will be initialized with the value 5. A parsable array is an array of parsable variables as shown in Figure 3(c). In order to provide more powerful semantics to array initialization, Flavor considers the parse size and initializer expressions as executed per each element of the array. The array size expression, however, is only executed once, before the parse size expression or the initializer expression.

In the example shown in Figure 3(c), the variable `A` is declared as an array of 2 integers. The first one is parsed with 3 bits and is expected to have the value 4, while the second is parsed with 5 bits and is expected to have the value 6. After the declaration, `var` is left with the value 6. This probably represents the largest deviation of Flavor's design from C++ and Java declarations. On the other hand it does provide significant flexibility in constructing sophisticated declarations in a very compact form, and it is also in line with the dynamic nature of variable declarations that Flavor provides.

Partial arrays: An additional refinement of array declaration is partial arrays. These are declarations of parsable arrays in which only a subset of the array needs to be declared (or, equivalently, parsed from or written to a bitstream). These are needed when the retrieved value indicates indirectly if further elements of the array should be parsed. Flavor introduces a double bracket notation for this purpose. The example in Figure 3(d) demonstrates its use. In the first line, we are declaring the 4^{th} element of the array `A` (array indices start from 0). The array size is unknown at this point, but of course it will be considered at least 4. In the second line, we are declaring a two-dimensional array, and in particular only its third row (assuming the first index corresponds to a row). The array indices can, of course, be expressions themselves. Partial arrays can only appear on the left-hand side of declaration and are not allowed in expressions.

3.2 Expressions and Statements

Flavor supports all of the C++ and Java arithmetic, logical and assignment operators; however, parsable variables cannot be used as lvalues. This ensures that they always represent the bitstream's content, and allow consistent operations for the translator-generated `get()` and `put()` methods that read and write, respectively, data according to the specified form. Refer to Section 4.1 for detailed information about these methods.

Flavor also supports all the familiar flow control statements: `if-else`, `for`, `do-while`, `while`, and `switch`. In contrast to C++ and Java, variable declarations are not allowed within the arguments of these statements (e.g., "`for(int i=0; ; );`" is not allowed). This is because in earlier versions of C++ the scope of this variable will be the enclosing one, while in Java it will be the enclosed one. To avoid confusion, we opted for the exclusion of both alternatives at the expense of a slightly more verbose notation. Scoping rules are discussed in detail in Section 3.4.

```
if (a == 1) {
  little int(16) b; // b is a 16 bit integer in little-endian byte ordering
else {
  little int(24) b; // b is a 24 bit integer in little-endian byte ordering
}
```

Figure 4: Example of a conditional expression.

Figure 4 shows an example of the use of these flow control statements. The variable `b` is declared with a parse size of 16 if `a` is equal to 1, and with a parse size of 24 otherwise. Observe that this construct would not be meaningful in C++ or Java as the two declarations would be considered to be in separate scopes. This is the reason why parsable variables need to obey slightly different scoping rules than regular variables. The way to approach this to avoid confusion is to consider that Flavor is designed so that these parsable variables have to be properly defined at the right time and position. All the rest of the code is there to ensure that this is the case. We can consider the parsable variable declarations as "actions" that our system will perform at the specified times. This difference, then, in the scoping rules becomes a very natural one.

3.3 Classes

```
class SimpleClass {
  int(3) a;
  unsigned int(4) b;
  if (b == 0b0010) {
    bit(8) c;
  }
}; // The trailling `;' is optional
```

(a)

```
class SimpleClass(int i[2]) {
  int(3) a = i[0];
  little int(24) b = i[1];
};

int(2) v[2];
SimpleClass sc(v);
```

(b)

Figure 5: Class. (a) A simple class declaration. (b) A simple class declaration with parameter types.

Flavor uses the notion of classes in exactly the same way as C++ and Java do. It is the fundamental structure in which object data are organized. Keeping in line with the support of both C++ and Java-style programming, classes in Flavor cannot be nested, and only single inheritance is supported. In addition, due to the declarative nature of Flavor, methods are not allowed (this includes constructors and destructors as well). Figure 5(a) shows an example of a simple class declaration with parsable member variables `a`, `b`, and `c`. The trailing '`;`' character is optional, accommodating both C++ and Java-style class declarations. The parsable variables will be present in the bitstream in the same order they are declared. After this class is defined, we can declare objects of this type as follows: "`SimpleClass sc;`".

A class is considered parsable if it contains at least one variable that is parsable. Declaration of parsable class variables can be prepended by the `aligned` modifier in the same way as parsable variables. Also, class member variables in Flavor do not require access modifiers (`public`, `protected`, or `private`). In essence, all such variables are considered public.

3.3.1 Parameter types

As Flavor classes cannot have constructors, it is necessary to have a mechanism to pass external information to a class. This is accomplished using *parameter types*. These act the same way as formal arguments in function or method declarations do. They are placed in parentheses after the name of the class. Figure 5(b) gives an example of a simple class declaration with parameter types. When declaring variables of parameter type classes, it is required that the actual arguments are provided in place of the formal ones as displayed in the figure. Of course the types of the formal and actual parameters must match. For arrays, only their dimensions are relevant; their actual sizes are not significant as they can be dynamically varying. Note that class types are allowed in parameter declarations as well.

3.3.2 Inheritance

As mentioned earlier, Flavor supports single inheritance so that compatibility with Java is maintained. Although Java can "simulate" multiple inheritance through the use of interfaces, Flavor has no such facility (it would be meaningless since methods do not exist in Flavor). However, for media representation purposes, we have not found any instance where multiple inheritance would be required, or even be desirable. It is interesting to note that all existing representation standards today are not truly object-based. The only exception, to our knowledge, is the MPEG-4 specification that explicitly addresses the representation of audio-visual objects. It is, of course, possible to describe existing structures in an object-oriented way but it does not truly map one-to-one with the notion of objects. For example, the MPEG-2 Video slices can be considered as separate objects of the same type, but of course their semantic interpretation (horizontal stripes of macroblocks) is not very useful. Note that containment formats like MP4 (MPEG-4 Systems file format) and Apple's QuickTime are more object-oriented, as they are composed of object-oriented structures called "atoms".

Derivation in C++ and Java is accomplished using a different syntax (`extends` versus '`:`'). Here we opted for the Java notation (also '`:`' is used for object identifier declarations as explained below). Unfortunately, it was not possible to satisfy both.

In Figure 6(a) we show a simple example of a derived class declaration. Derivation from a bitstream representation point of view means that `B` is an `A` with some additional information. In other words, the behavior would be almost identical if we just copied the statements between the braces in the declaration of `A` in the beginning of `B`. We say "almost" here because scoping rules of variable declarations also come into play, as discussed in Section 3.4.

Note that if a class is derived from a parsable class, it is also considered parsable.

3.3.3 Polymorphic parsable classes

The concept of inheritance in object-oriented programming derives its power from its capability to implement polymorphism. In other words, the capability to use a derived object in a place where an object of the base class is expected. Although the mere structural organization is useful as well, it could be accomplished equally well with containment (a variable of type `A` is the first member of `B`).

```
class A {              class A:int(1) id = 0 {
  int(2) a;              int(2) a;
}                      }

class B extends A {    class B extends A:int(1) id = 1 {
  int(3) b;              int(3) b;
}                      }
     (a)                            (b)
```

Figure 6: Inheritance. (a) Derived class declaration. (b) Derived class declaration with object identifiers.

Polymorphism in traditional programming languages is made possible via vtable structures, which allow the resolution of operations during run-time. Such behavior is not pertinent for Flavor, as methods are not allowed.

A more fundamental issue, however, is that Flavor describes the bitstream syntax: the information with which the system can detect which object to select *must be present in the bitstream*. As a result, traditional inheritance as defined in the previous section *does not* allow the representation of polymorphic objects. Considering Figure 6(a), there is no way to figure out, by reading a bitstream, if we should read an object of type `A` or type `B`.

Flavor solves this problem by introducing the concept of *object identifiers* or IDs. The concept is rather simple: in order to detect which object we should parse/generate, there must be a parsable variable that will identify it. This variable must have a different expected value for any class derived from the originating base class, so that object resolution can be uniquely performed in a well-defined way (this can be checked by the translator). As a result, object ID values must be constant expressions and they are always considered constant, i.e., they cannot be redeclared within the class.

In order to signify the importance of the ID variables, they are declared immediately after the class name (including any derivation declaration) and before the class body. They are separated from the class name declaration using a colon (':'). We could rewrite the example of Figure 6(a) with IDs as shown in Figure 6(b). Upon reading the bitstream, if the next 1 bit has the value 0, an object of type `A` will be parsed; if the value is 1, then an object of type `B` will be parsed. For output purposes, and as will be discussed in Section 4, it is up to the user to set up the right object type in preparation for output.

The name and the type of the ID variable is irrelevant, and can be anything that the user chooses. It cannot, however, be an array or a class variable (only built-in types are allowed). Also, the name, type and parse size must be identical between the base and derived classes. However, object identifiers are not required for all derived classes of a base class that has a declared ID. In this case, only the derived classes with defined IDs can be used wherever the base class can appear. This type of polymorphism is already used in the MPEG-4 Systems specification, and in particular the Binary Format for Scenes (BIFS) [22]. This is a VRML-derived set of nodes that represent objects and operations on them, thus forming a hierarchical description of a scene.

```
class slice : aligned bit(32) slice_start_code = 0x00000101 .. 0x000001AF {
  ...
}
```

Figure 7: Declaration of a class with an ID range.

The ID of a class is also possible to have a range of possible values which is specified as `start_id` ... `end_id`, inclusive of both bounds. See Figure 7 for an example.

3.4 Scoping Rules

The scoping rules that Flavor uses are identical with C++ and Java with the exception of parsable variables. As in C++ and Java, a new scope is introduced with curly braces ({}). Since Flavor does not have functions or methods, a scope can either be the global one or a scope within a class declaration. Note that the global scope cannot contain any parsable variable, since it does not belong to any object. Only global variables that are constant are allowed.

Within a class, all parsable variables are considered as class member variables, regardless of the scope they are encountered in. This is essential in order to allow conditional declarations of variables which will almost always require that the actual declarations occur within compound statements (see Figure 4). Non-parsable variables that occur in the top-most class scope are also considered class member variables. The rest live within their individual scopes.

```
class A {                          class B {
  int i = 1;                         A a;
  int(2) a;                          a.j = 1;          // Error, j not a class member
  if (a == 2) {                      if (a.i == 4) {    // Ok
    int j = i;                         int j = a.a + 1; // Ok
    int i = 2; // Ok, hides i          j = a.i + 2;     // Ok
    int a;     // Error, hides a       int(3) b;
  }                                  }
}                                  }
```

Figure 8: A scoping rules example.

This distinction is important in order to understand which variables are accessible to a class variable that is contained in another class. The issues are illustrated in Figure 8. Looking at the class `A`, the initial declaration of `i` occurs in the top-most class scope; as a result `i` is a class member. The variable `a` is declared as a parsable variable, and hence it is automatically a class member variable. The declaration of `j` occurs in the scope enclosed by the `if` statement; as this is not the top-level scope, `j` is not a class member. The following declaration of `i` is acceptable; the original one is hidden within that scope. Finally, the declaration of the variable `a` as a non-parsable would hide the parsable version. As parsable variables do not obey scoping rules, this is not allowed. Looking now at the declaration of the class `B` which contains a variable of type `A`, it becomes clear which variables are available as class members.

In summary, the scoping rules have the following two special considerations. Parsable variables do not obey scoping rules and are always considered class members. Non-parsable variables obey the standard scoping rules and are considered class members only if they are at the top-level scope of the class.

Note that parameter type variables are considered as having the top-level scope of the class. Also, they are not allowed to hide the object identifier, if any.

3.5 Maps

Up to now, we have only considered fixed-length representations, either constant or parametric. A wide variety of representation schemes, however, rely heavily on entropy coding, and in particular Huffman codes [23] (e.g., Group 3 [24] and Group 4 [25] fax, MPEG-1 [4], MPEG-2 [5], and etc.). These are variable-length codes (VLCs) which are instantaneously decodable (no codeword is the prefix of another). Flavor provides extensive support for variable-length coding through the use of maps that are declarations of tables in which the correspondence between codewords and thier values is described.

Figure 9(a) gives a simple example of a map declaration. The `map` keyword indicates the declaration of a map named `A`. The declaration also indicates that the map converts from bit string values to values of type `int`. The type indication can be a fundamental type, a class type, or an array. Map declarations can only occur in global scope. As a result, an array declaration will have to have a constant size (no non-constant variables are visible at this level). After the map is properly declared, we can define parsable variables that use it by indicating the name of the map where we would put the parse size expression as follows: "`int(A) a;`". As we can see, the use of VLCs is essentially identical to fixed-length variables. All the details are hidden away in the map declaration.

```
map A(int) {         // The output type of a map is defined in a class
  0b0, 1,            class YUVblocks {
  0b01, 2,             unsigned int Yblocks;
  0b001, 3             unsigned int Ublocks;
}                      unsigned int Vblocks;
    (a)              }

                     // A table that relates the chroma format with
                     // the number of blocks per signal component
map A(int) {         map BlocksPerComponent (YUVblocks) {
  0b0, 1,              0b00, {4, 1, 1},   // 4:2:0
  0b01, 2,             0b01, {4, 2, 2},   // 4:2:2
  0b001, int(5)        0b10, {4, 4, 4}    // 4:4:4
}                    }
    (c)                              (b)
```

Figure 9: Map. (a) A simple map declaration. (b) A map with defined output type. (c) A map declaration with extension.

The map contains a series of entries. Each entry starts with a bit string that declares the codeword of the entry followed by the value to be assigned to this codeword. If a complex type is used for the mapped value, then the values have to be enclosed in curly braces. Figure 9(b) shows the definition of a VLC table with a user-defined class as output type. The type of the variable has to be identical to the type returned from the map. For example, using the "`YUVblocks(BlocksPerComponent) chroma_format;`" definition, we can access a particular value of the map using the construct: "`chroma_format.Ublocks`".

As Huffman 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. To accommodate these as well as more sophisticated constructs, Flavor allows the use of parsable type indications in map values. This means that, using the example in Figure 9(a), we can write the example in Figure 9(c), which indicates that, when the bit string `0b001` is encountered in the bitstream, the actual return value for the map will be obtained by parsing 5 more bits. The parse size for the extension can itself be a map, thus allowing the cascading of maps in sophisticated ways. Although this facility is efficient when parsing, the bitstream generation operation can be costly when complex map structures are designed this way. None of today's specifications that we are aware of require anything beyond a single escape code.

Section 4.6 provides the basic information on how the translator generates map processing code.

3.6 Built-In Operators

Operators are built-in functions that are made available to the Flavor programmer in order to facilitate certain frequently appearing data manipulations. These operators are the only functions that are available in the current version of Flavor.

3.6.1 lengthof()

The `lengthof()` operator is used to obtain the parsed length of a parsable variable. The general syntax is `lengthof(var)`, where `var` must be a parsable variable of any type that has been previously declared. The result of the operator is treated as an integer. Since parsable variables can be declared more than once, the operator considers only the last instance of the variable that has been declared (parsed from the bitstream). Figure 10 shows examples of using of the operator. In Figure 10(b), we declare an one-dimensional array `A` with five elements. The variable `b` is set with the length of the last (4^{th}) element. It is easy to see that the length of this element is going to be 5.

```
int(5) a = 3;                int i = 1;
...                          int(i++) A[5];
int(3) b = lengthof(a);      int b = lengthof(A[4]);
      (a)                               (b)
```

Figure 10: Examples of using the `lengthof()` operator.

In a declaration such as "`aligned class A{...};`", the bits skipped for alignment are not accounted for by the `lengthof()` operator. This is true for simple variables as well. These bits, however, are counted for the enclosing class.

3.6.2 isidof()

The `isidof()` operator is used to check if the value of a varible is among the IDs of a polymorphic class. The general syntax is `isidof(cname, id)`, where `cname` is the name of a polymorphic class that has been previously declared, and `id` is the name of a simple variable. The result of the operator is treated as an integer, and it is 1 if the value of `id` is among the IDs of `cname` or 0, otherwise.

This operator was introduced to accommodate a coding structure in which the syntax was expressed as: pass as many objects of a particular type as you can from the bitstream. This corresponds to examining the following bits on the bitstream and, if they correspond to an object of the given type, parsing it; otherwise the syntax would continue to the next construct. Without this operator, the programmer would have to explicitly construct a `switch` statement or a series of `if-then-else` statements, checking against all IDs of the class. This not only would be tiresome, but would also be a source of errors if one of the IDs were not included.

Figure 11 shows an example of the use of this operator. In the figure, we declare an abstract class `A`, with presumably a number of derived classes. In the main class (`Main`), the `while` loop examines the next 8 bits of the bitstream and if they correspond to an ID of one of the classes derived from `A` then the object is parsed; if not, the code continues.

```
abstract class A : int(8) id = 0 {    class Main {
...                                     int i;
}                                       int(8)* id;
                                        while (isidof(A, id)==1) {
class B extends A : int(8) id = 1 {       A a[[i++]];
...                                       int(8)* id;
}                                       }
...                                   }
```

Figure 11: An example of using the `isidof()` operator.

3.6.3 skipbits()

The `skipbits(n)` operator is used to skip `n` bits from the input bitstream, where `n` can be an integer constant or a non-constant variable of type compatible to `int`; in the case of outputting bits, zeroes are added. This operator is useful for skipping information that is not pertinent to an application. For example, an MPEG-2 Systems stream (a transport stream) can consist of multiple elementary streams (e.g., one video stream and three audio streams – for three different languages), each packetized into packets of 188 bytes. With the size of the packets known, the packets of unwanted streams can be simply skipped using the `skipbits()` operator.

3.6.4 nextbits()

The `nextbits()` operator is used as a substitute for look-ahead parsing. The general syntax is as follows: `nextbits([aligned(length),] [(big|little),] [(signed|unsigned),] n)`, where `n` determines the number of bits to look-ahead read. The operator, by default, returns the signed integer representation of the next `n` bits in big-endian byte-ordering. The optional `aligned` parameter can be used exactly in the same way as the `aligned` modifier is used in the parsable variable declarations. The (`big` or `little`) and (`signed` or `unsigned`) optional parameters can be used to indicate the integer representation of the `n` bits read. Using the operator instead of the look-ahead parsable variable avoids declaring an extra variable.

3.6.5 nextcode()

The `nextcode()` operator, with the syntax – `nextcode([aligned(length),] code)`, is used to search for a given `code` where the `code` is a constant binary, octal, or hexadecimal literal; only the literals that also convey their length are allowed. For example, in the MPEG-2 Video specification, the `0x00000100` code is used to indicate the start of a picture (frame) and the "`nextcode(aligned(8), 0x00000100);`" statement skips all bits upto but excluding the next `0x00000100` code, where the code is byte aligned. When outputting bits, zeroes are added until the next byte-aligned position.

3.6.6 numbits()

The `numbits()` operator is used to obtain the total number of bits read/written so far.

4 The Flavor Translator

Designing a language like Flavor would be an interesting but academic exercise, unless it was accompanied by software that can put its power into full use. We have developed a translator that evolved concurrently with the design of the language. When the language specification became stable, the translator was completely rewritten. The most recent release is publicly available for downloading at the Flavor web site: `http://flavor.sourceforge.net`.

4.1 Run-Time API

The translator reads a Flavor source file (`.fl`) and, depending on the language selection flag, it generates a pair of `.h` and `.cpp` files (for C++) or a set of `.java` files (for Java). In the case of C++, the `.h` file contains the declarations of all Flavor classes as regular C++ classes and the `.cpp` file contains the implementations of the corresponding class methods (`get()` and `put()`). In the case of Java, each `.java` file contains the declaration and implementation of a single Flavor class. In both cases, the `get()` method is responsible for reading a bitstream and loading the class variables with their appropriate values, while the `put()` method does the reverse. All the members of the classes are declared `public`, and this allows direct access to desired fields in the bitstream.

The translator makes minimal assumptions about the operating environment for the generated code. For example, it is impossible to anticipate all possible I/O structures that might be needed by applications (network-based, multi-threaded, multiple buffers, etc.). Attempting to provide a universal solution would be futile. Thus, instead of having the translator directly output the required code for bitstream I/O, error reporting, tracing, etc., a run-time library is provided. With this separation, programmers have the flexibility of replacing parts of, or the entire library with their own code. The only requirement is that the customized code provides an identical interface to the one needed by the translator. This interface is defined in a pure virtual class called `IBitstream`. Deriving from the provided `IBitstream` class will ensure compatibility with the translator, and as the source code for the library is included in its entirety, customization can easily be performed. The Flavor package also provides information on how to rebuild the library, if needed.

The run-time library includes the `Bitstream` class that is derived from the `IBitstream` interface, and provides basic bitstream I/O facilities in terms of reading or writing bits from a binary file. An `IBitstream` reference is passed as an argument to the `get()` and `put()` methods so that any bitstream I/O class derived from the `IBitstream` interface can be used. For example, we have created a new bitstream I/O class for accessing MP3 files (`BuffBitstream`), which it is currently available in the Flavor web site. The included `Bitstream` class only supports sequential bitstream access, as all the parsed bits are discarded; however, a buffered I/O class is needed to properly process MP3 files, and we have derived the `BuffBitstream` I/O class that buffers parsed bits until they are no longer needed.

If parameter types are used in a class, then they are also required arguments in the `get()` and `put()` methods as well. The translator also requires that a function is available to receive calls when expected values are not available or VLC lookups fail. The function name can be selected by the user; a default implementation (`flerror`) is included in the run-time library.

For efficiency reasons, Flavor arrays are converted to fixed size arrays in the translated code. This is necessary in order to allow developers to access Flavor arrays without needing special techniques. Whenever possible, the translator automatically detects and sets the maximum array size; it can also be set by the user using a command-line option. Finally, the run-time library only allows parse sizes of up to the native integer size of the host processor (except for double values). This enables fast implementation of bitstream I/O operations.

For parsing operations, the only task required by the programmer is to declare an object of the class type at hand, and then call its `get()` method with an appropriate bitstream. While the same is also true for the `put()` operation, the application developer must also load all class member variables with their appropriate values before the call is made.

4.2 Include and Import Directives

In order to simplify the source code organization, Flavor supports `%include` and `%import` directives. These are the mechanisms to combine several different source code files into one entity, or to share a given data structure definition across different projects.

4.2.1 Include Directive

The statement – `%include "file.fl"` – will include the specified `.fl` file in the current position and will flag all of its content so that no code is generated. Figure 12 displays a `.fl` file (`other.fl`) that is included by another `.fl` file (`main.fl`). The `other.fl` file contains the definition of the constant `a`. The inclusion makes the declaration of the `a` variable available to the `main.fl` file. In terms of the generated output, the main and included files each keep their corresponding implementations. The generated C++ code maintains this partitioning, and makes sure that the main file includes the C++ header file of the included Flavor file. Similarly, when generating the Java code, only the `.java` files corresponding to the currently processed Flavor file are generated.

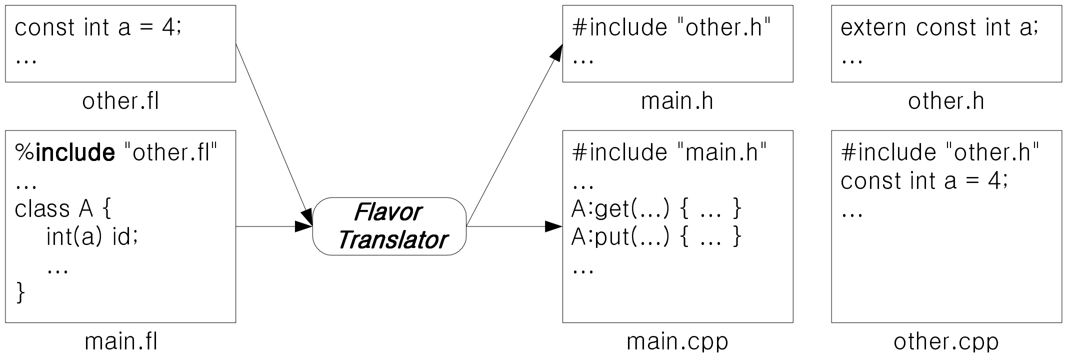

Figure 12: The `other.fl` file is included by the `main.fl` file, and the Flavor translator generates the `main.h` and `main.cpp` files from the `main.fl` file. The `other.h` and `other.cpp` files are generated separately from the `other.fl` file.

The `%include` directive is useful when data structures need to be shared across modules or projects. It is similar in spirit to the use of the C/C++ preprocessor `#include` statement in the sense that it is used to make general information available at several different places in a program. Its operation, however, is different as Flavor's `%include` statement does not involve code generation for the included code. In C/C++, `#include` is equivalent to copying the included file in the position of the `#include` statement. This behavior is offered in Flavor by the `%import` directive. Note that the Java `import` statement behaves more like Flavor's `%include` statement, in that no code generation takes place for the imported code.

4.2.2 Import Directive

The `%import` directive behaves similarly to the `%include` directive, except that full code is generated for the imported file by the translator, and no C++ `#include` statement is used. This behavior is identical to how a C++ preprocesor `#include` statement would behave in Flavor.

Consider the example shown in Figure 13, which uses an `%import` directive rather than an `%include` one. Here the generated code includes the C++ code corresponding to the imported `.fl` file. Therefore, using the `%import` directive is exactly the same as just copying the code in the imported `.fl` file and pasting it in the same location as the `%import` statement is specified. The translator generates the Java code in the same way.

4.3 Pragma Statements

Pragma statements are used as a mechanism for setting translator options from inside a Flavor source file. This allows modification of translation parameters (set by the command-line options) without modifying the makefile that builds the user's program, but more importantly, it allows very fine control on which translation options are applied to each class, or even variable. Almost all command-line options have pragma equivalents. The ones excluded were not considered useful for specification within a source file.

Pragma statements are introduced with the `%pragma` directive. It can appear wherever a statement or declaration can. It can contain one or more settings, separated by commas, and it

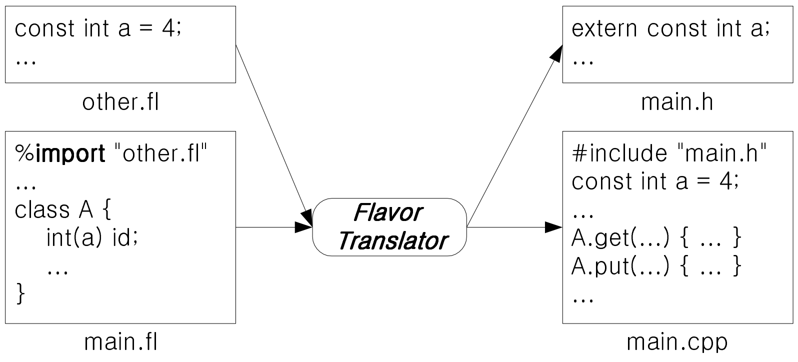

Figure 13: The Flavor translator generates the `main.h` and `main.cpp` files from the `main.fl` file, and the generated file includes the code corresponding to the `other.fl` file imported by the `main.fl` file.

cannot span more than one line. After a setting is provided, it will be used for the remainder of the Flavor file, unless overridden by a different pragma setting. In other words, pragma statements do not follow the scope of Flavor code. A pragma that is included in a class will affect not only the class where it is contained, but all classes declared after it. An example is provided in Figure 14.

```
// Activate both put and get, generate tracing code, and set array size to 128
%pragma put, get, trace, array=128
class Example {
  %pragma noput                 // No put() method needed

  unsigned int(10) length;
  %pragma array=1024            // Switch array size to 1024
  char(3) data[length];

  %pragma array=128             // Switch array size back to 128
  %pragma trace="Tracer.trace"  // Use custom tracer
}
// The above settings are still active here!
```

Figure 14: Some examples of using pragma statements to set the translator options at specific locations.

In this example, we start off setting the generation of both `get()` and `put()` methods, enabling tracing and setting the maximum array size to 128 elements. Inside the `Example` class, we disable the `put()` method output. This class reads a chunk of data, which is preceded by its size (`length`, a 10-bit quantity). This means that the largest possible buffer size is 1024 elements. Hence for the `data` array that immediately follows, we set the array size to 1024, and then switch it back to the default of 128. Finally, at the end of the class we select a different tracing function name; this function is really a method of a class, but this is irrelevant for the translator. Since this directive is used when the `get()` method code is produced, it will affect the entire class despite the fact that it is declared at its end.

Note that these pragma settings remain in effect even after the end of the `Example` class.

4.4 Verbatim Code

In order to further facilitate integration of Flavor code with C++/Java user code, the translator supports the notion of verbatim code. Using special delimiters, code segments can be inserted in the Flavor source code, and copied verbatim at the correct places in the generated C++/Java file.

This allows, for example, the declaration of constructors/destructors, user-specified methods, pointer member variables for C++, etc. Such verbatim code can appear wherever a Flavor statement or declaration is allowed.

The delimiters `%{` and `%}` can be used to introduce code that should go to the class declaration itself (or the global scope). The delimiters `%g{` and `%g}`, and `%p{` and `%p}` can be used to place code at exactly the same position they appear in the `get()` and `put()` methods, respectively. Finally, the delimiters `%*{` and `%*}` can be used to place code in both `get()` and `put()` methods. To place code specific to C++ or Java, `.c` or `.j` can be placed before the braces in the delimiters, respectively. For example, a verbatim code to be placed in the `get()` method of the Java code will be delimited with `%g.j{` and `%g.j}`.

```
class GIF87a {
  char(8) GIFsignature[6] = "GIF87a"; // The GIF signature

  %g{ print(); %g}

  // A screen descriptor
  ScreenDescriptor sd;

  // One or more image descriptors
  do {
    unsigned int(8) end;

    // We found an image descriptor
    if (end == ',') ImageDescriptor id;

    // We found an extension block
    if (end == '!') ExtensionBolck eb;

    // Everything else is ignored and ';' is the end-of-data marker
  } while (end != ';');

  %.c{ void print() { ... } %.c}
}
```

Figure 15: A simple Flavor example: the GIF87a header. The usage of verbatim code is illustrated.

The Flavor package includes several samples on how to integrate user code with Flavor-generated code. Figure 15 shows a simple example that reads the header of a GIF87a file and prints its values. The print statement which prints the values of the various elements is inserted as verbatim code in the syntax (within `%g{` and `%g}` markers, since the code should go in the `get()` method). The implementation of the print method for C++ code is declared within `%.c{` and `%.c}`, and for Java, the corresponding implemenation can be defined within `%.j{` and `%.j}`. The complete sample code can be found in the Flavor package.

4.5 Tracing Code Generation

We also included the option to generate bitstream tracing code within the `get()` method. This allows one to very quickly examine the contents of a bitstream for development and/or debugging purposes by creating a dump of the bitstream's content. With this option, and given the syntax of a bitstream described in Flavor, the translator will automatically generate a complete C++/Java program that can verify if a given bitstream complies with that syntax or not. This can be extremely useful for codec development as well as compliance testing. Figure 16 displays the output generated (with the Flavor-generated tracing code) when parsing a GIF87a file. Each line lists a parsable variable or a class with its position in the bitstream, parse size (in bits), corresponding value and name.

```
At Bit  Size    Value    Description
    0:                   begin GIF87a
    0:     8       47    GIFsignature[0] (G)
    8:     8       49    GIFsignature[0] (I)
   16:     8       46    GIFsignature[0] (F)
   24:     8       38    GIFsignature[0] (8)
   32:     8       37    GIFsignature[0] (7)
   40:     8       61    GIFsignature[0] (a)
   48:                   processing ScreenDescriptor sd
   48:                   begin ScreenDescriptor
   48:    16    00 9A    width (154)
   64:    16    00 70    height (112)
   80:     1       01    M (1)
   81:     3       07    cr (7)
   84:     1       00    marker (0)
   85:     3       07    pixel (7)
   88:     8       00    background (0)
   ...
```

Figure 16: A sample output generated using the Flavor-generated tracing code

4.6 Generating Code for Map Processing

Map processing is one of the most useful features of Flavor, as hand-coding VLC tables is tedious and error prone. Especially during the development phase of a representation format, when such tables are still under design, full optimization within each design iteration is usually not performed. By using the translator, such optimization is performed at zero cost. Also, note that maps can be used for fixed-length code mappings just by making all codewords have the same length. As a result, one can very easily switch between fixed and variable-length mappings when designing a new representation format.

When processing a map, the translator first checks that it is instantaneously decodable, i.e., no codeword is the prefix of another. It then constructs a class declaration for that map, which exposes two methods: `getvlc()` and `putvlc()`. These take as arguments a reference to the `IBitstream` interface as well as a pointer to the return type of the map. The `getvlc()` method is responsible for decoding a map entry and returning the decoded value, while the `putvlc()` method is responsible for the output of the correct codeword. Note that the defined class does not perform any direct bitstream I/O itself, but uses the services of the `Bitstream` or a similar class instead, which is derived from the `IBitstream` interface. This ensures that a user-supplied bitstream I/O library will be seamlessly used for map processing.

The code generated by Flavor uses a hybrid method for decoding VLCs [26]. The method uses nested `switch` statements, instead of lookup tables (LTs). The benefit of using this method is that only complete matches require `case` statements, while all partial matches can be grouped into a single `default` statement (that, in turn, introduces another `switch` statement). The method is further optimized by ordering the `case` statements in terms of the length of their codewords. As shorter lengths correspond to higher probabilities, this minimizes the average number of comparisons per codeword. For more detailed information, refer to [26].

5 Concluding Remarks

Flavor's design was motivated by our belief that content creation, access, manipulation and distribution will become increasingly important for end-users and developers alike. New media representation forms will continue to be developed, providing richer features and more functionalities for end-users. In order to facilitate this process, it is essential to bring syntactic description on par with modern software development practices and tools. Flavor can provide significant benefits in the area of media representation and multimedia application development at several levels.

First, it 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. This by itself is a substantial advantage for defining specifications, as a considerable amount of time is spent to ensure that such specifications are unambiguous and bug-free.

Second, a formal media representation language immediately leads to the capability of automatically generating software tools, ranging from bitstream generators and verifiers, as well as a substantial portion of an encoder or decoder.

Third, it allows immediate access to the content by any application developer, for such diverse use as editing, searching, indexing, filtering, etc. With appropriate translation software, and a bitstream representation written in Flavor, obtaining access to such content is as simple as cutting and pasting the Flavor code from the specification into an ASCII file, and running the translator.

Flavor, however, does not provide facilities to specify how full decoding of data will be performed as it only addresses bitstream syntax description. For example, while the data contained in a GIF file can be fully described by Flavor, obtaining the value of a particular pixel requires the addition of LZW decoding code that must be provided by the programmer. In several instances, such access is not necessary. For example, a number of tools have been developed to do automatic indexing, searching and retrieval of visual content directly in the compressed domain for JPEG and MPEG content (see [27, 28]). Such tools only require parsing of the coded data so that DCT [29] coefficients are available, but do not require full decoding. Also, new techniques, such as MPEG-7, will provide a wealth of information about the content without the need to decode it. In all these cases, parsing of the compressed information may be the only need for the application at hand.

Finally, Flavor can also be used to redefine the syntax of content in both forward and backward compatible ways. The separation of parsing from the remaining encoding/decoding operations allows its complete substitution as long as the interface (the semantics of the previously defined parsable variables) remains the same. Old decoding code will simply ignore the new variables, while newly written encoders and decoders will be able to use them. Use of Java in this repect is very useful; its capability to download new class definitions opens the door for such downloadable content descriptions that can accompany the content itself (similar to self-extracting archives). 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.

Acknowledgements

The authors gratefully acknowledge Olivier Avaro (France Telecom), Carsten Herpel (Thomson Multimedia), and Jean-Claude Dufourd (ENST) for their contributions in the Flavor specification during the development of the MPEG-4 standard. We would also like to acknowledge Yuntai Kyong and Yihan Fang, who helped to implement early versions of the Flavor translator.

References

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

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

[3] ISO/IEC 10918 International Standard (JPEG). *Information technology – Digital compression and coding of continuous-tone still images*, 1994.

[4] 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.

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

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

[7] A. Eleftheriadis. *mkvlc: A Variable Length Code Table Compiler.* Columbia University, 1994.

[8] Y. Fang and A. Eleftheriadis. "A Syntactic Framework for Bitstream-Level Representation of Audio-Visual Objects". In *IEEE Int. Conf. on Image Processing*, Proceedings, pages II.426–II.432, 1996.

[9] A. Eleftheriadis. *A Syntactic Description Language for MPEG-4.* ISO/IEC JTC1/SC29/WG11 M546, 1995.

[10] J. Levine, T. Mason, and D. Brown. *lex & yacc.* O'Reilly, 2^{nd} edition, 1992.

[11] IETF RFC 1014. *XDR: External Data Representation Standard*, 1987.

[12] IETF RFC 1832. *XDR: External Data Representation Standard*, 1995.

[13] R. Stevens. *Unix Network Programming*, volume 2. Prentice Hall, 2^{nd} edition, 1999.

[14] http://www.omg.org. *CORBA IDL.*

[15] ISO/IEC 8824. *Abstract Syntax Notation One (ASN.1)*, 2002.

[16] ISO/IEC 8825. *ASN.1 Encoding Rules*, 2002.

[17] S. Devillers and M. Caprioglio. *Bitstream Syntax Description Language (BSDL).* ISO/IEC JTC1/SC29/ WG11 M7433, 2001.

[18] S. Devillers, M. Amielh, and T. Planterose. *Bitstream Syntax Description Language (BSDL).* ISO/IEC JTC1/SC29/WG11 M8273, 2002.

[19] ISO/IEC JTC1/SC29/WG11 N4801, Fairfax. *MPEG-21 Overview v.4*, May 2002.

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

[21] D. Hong and A. Eleftheriadis. "XFlavor: Bridging Bits and Objects in Media Representation". In *IEEE Int. Conf. on Multimedia and Expo*, Proceedings, pages 773–776, 2002.

[22] ISO/IEC 14496-1 International Standard. *Information technology – Coding of audio-visual objects – Part 1: Systems*, 2001.

[23] D. A. Huffman. “A Method for the Construction of Minimum Redundancy Codes”. *Proceedings of the IRE*, 40:1098–1101, 1952.

[24] CCITT (ITU Recommendation T.4). *Standardization of Group 3 Facsimile Apparatus for Document Transmission*, 1980, amended in 1984 and 1988.

[25] CCITT (ITU Recommendation T.11). *Facsimile Coding Schemes and Coding Control Functions for Group 4 Facsimile Apparatus*, 1984, amended in 1988.

[26] Y. Fang and A. Eleftheriadis. “Automatic Generation of Entropy Coding Programs Using Flavor”. In *IEEE Workshop on Multimedia Signal Processing*, Proceedings, pages 341–346, 1998.

[27] S. W. Smoliar and H. Zhang. “Content-Based Video Indexing and Retrieval”. *IEEE Multimedia Magazine*, 1994.

[28] J. R. Smith and S. F. Chang. “VisualSEEk: A Fully Automated Content-Based Image Query System”. In *ACM Int. Conf. on Multimedia*, Proceedings, 1996.

[29] K. R. Rao and P. Yip. *Discrete Cosine Transform Algorithms, Advantages, Applications.* Academic Press, 1990.