Specification

Chapter 1

BFlavor specification

With the BFlavor specification, it is possible to describe the syntax structure of coding format in a C++-like manner. Note that BFlavor is built on the Formal Language for Audio-Visual Object Representation, extended with XML features (XFlavor). Therefore, this specification is based on the XFlavor specification which can be found at http://flavor.sourceforge.net.

1.1 Variables

Regular variable declarations can be used in BFlavor in the same way as in C++ or Java. Besides regular variable declarations, it is also possible to use parsable variables in BFlavor. Parsable variables are the core of BFlavor's design; because their value is fetched from the bitstream itself, it are parsable variables that define the bitstream syntax. Note that parsable variables, in contrast with regular variables, will occur in the resulting bitstream description. Parsable variables include a parse length specification immediately after their type declaration, as shown in the following example.

```
int(5) var;
```

In the above example, the length of the parsable variable `var` is defined to be five bits. The length must be an integer constant lower or equal to 64. In general, the parse size expression must be a non-negative value. When the parsing process reaches this declaration, five bits will be read from the bitstream. The value of these bits will be assigned to the variable `var`.

Note that variables can only occur inside a class (discussed in Section 1.6). Hence, the use of global variables in a BFlavor code is prohibited.

Names

Variable names follow the C++ and Java conventions (i.e., variable names must start with a letter or an underscore). The keywords that are used in C++ and Java are also considered reserved in BFlavor.

Bitstream validation

Very often, certain syntax elements in the specification of a coding format will have fixed values (markers, start codes, reserved bits, etc.). These syntax elements can be checked by assigning the fixed values to their corresponding parsable variable:

```
int(5) var = 7;
```

Additionally, a parsable variable may be allowed to have a range of expected values:

```
int(5) var = 7 .. 12;
```

As both the parse size and the initial value can be arbitrary expressions, we should note that the order of evaluation is the parse expression first, followed by the initializing expression. During the parsing process, parsable variables containing fixed values will be checked. When the value in the bitstream does not matches the value (or does not fit in the range) given in the BFlavor code, the parser will notify this inconsistency. Note that this will not stop the parsing process.

1.2 Built-in functions

BFlavor provides five built-in functions that can assist in describing the high-level syntax of a coding format.

1.2.1 nextbits()

A look-ahead mechanism is provided by the `nextbits()` function. It takes as parameter the amount of bits one wants to look forward into the bitstream. The value of these bits is returned by this function. The general syntax is as follows:

```
nextbits(n);
```

`n` determines the number of bits to look ahead in the bitstream. This function returns the integer representation of the next `n` bits in big-endian byte ordering.

1.2.2 getcontext()

The `getcontext()` function is used to retrieve information of context classes (which will be discussed further in this specification). Context classes are useful when information is needed that is already parsed, but that is located in another class. The general syntax of the `getcontext()` function is as follows:

```
getcontext("classname", index, $path.$to.$variable);
```

The first parameter of the `getcontext()` function is a string which contains the name of the context class. The second parameter is an expression which represents the index value of the index element of the context class. The third parameter is the variable of the context class from which we want to know the value. Because this variable will not be defined in the class where we use this operator, a special notation for this kind of variables has to be used. This is done by prefixing the variable with `$`. A context class can contain other classes. These classes are automatically saved in the context. In the example, we see that the class `classname` contains a class variable `path`. The class `path` contains a class variable `to` which contains the parsable variable `variable`. The variable `variable` can accessed via `$path.$to.$variable`.

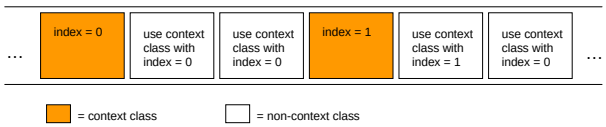

Figure 1.1: Context classes in a bitstream.

1.2.3 numbits()

This function returns the amount of bits that are already parsed since the start of the parsing process. Consider the following example:

```
int nrOfBitsParsed = numbits();
```

In this example, the non-parsable variable `nrOfBitsParsed` will contain the amount of bits that is already parsed from the bitstream since the start of the parsing process.

1.2.4 align()

The `align()` function reads bits from the bitstream until it reached a byte-aligned position. The value of the padding bits will be added to the resulting bitstream description. The following example illustrates the use of this function.

```
//start from byte-aligned position in the bitstream
bit(2) e1;
align();
//bitstream is back on a byte-aligned position
```

In this example, the `align()` function will parse six bits in order to set the bitstream back on byte-aligned position.

1.2.5 skipbits()

Skipping bits from the bitstream is possible with the `skipbits()` function. It takes as parameter the amount of bits that have to be skipped. Note that the skipped bits will be lost: they will not appear in the resulting bitstream description. The following example illustrates this function:

```
bit(3) e1;
skipbits(4);
bit(1) e2;
```

1.3 Datatypes

The variables occurring in a class can have built-in datatypes or user-defined datatypes.

1.3.1 Built-in datatypes

The following built-in datatypes are supported by BFlavor.

- **char, int, float, double**: BFlavor supports the common subset of C++ and Java built-in or fundamental types. This includes `char`, `int`, `float`, and `double`. Examples are shown below.

  ```
  char(8) letter;
  int(16) integer;
  ```

- **bit**: In addition, BFlavor defines a type called `bit`. The type `bit` is to accommodate bit string variables. An example is shown below.

  ```
  bit(2) twobits;
  ```

- **hexBinary**: the hexBinary datatype is used to describe a syntax element in a hexadecimal manner. The length of such elements is given in terms of bytes. This datatype is typically used for the description of start codes. The use of this datatype is as follows:

  ```
  hexBinary(3) three_bytes_start_code;
  ```

- **byteRange**: the byteRange datatype is used for referring to a particular bitstream segment. For instance, this is needed when a part of the bitstream syntax is not relevant to appear in the resulting bitstream description. Only the start and length of the bitstream segment are written into the bitstream description in order to correctly reconstruct the bitstream. The length of the bitstream segment that has to be referred (and not be described) can be specified in two different ways. First, it is possible to indicate that the bitstream segment stops when a particular bit string (or range of bit strings) is reached. Note that the length of the bit string must be specified (in terms of bytes) together with the value of the bit string. Second, the length of the bitstream segment can simply be given in terms of bytes. The two possibilities are illustrated below.

  ```
  //a byteRange variable searching for two codes in the bitstream:
  //0x000001 and 0x000000
  byteRange(3) payload1[2] = {0x000000, 0x000001};
  ```

```
//a byteRange variable searching for codes in the bitstream
//within the range of 0x000000 and 0x000001
byteRange(3) payload2 = 0x000000 .. 0x000001;

//a byteRange variable containing a length of 20 bytes
byteRange(20) payload3;
```

BFlavor does not support pointers, references, casts, or C++ operators related to pointers. It also does not support structures or enumerations. Finally, variables of type float, double are only allowed to have a parse size equal to the fixed size that their standard representation requires.

1.3.2 User-defined datatypes

BFlavor offers the possibility to create new datatypes besides the already existing built-in datatypes. A class that represents a user-defined datatype has to extend the `Encoded` base class. The class `Encoded` is a simple built-in base class from which other classes may be derived. It specifies a well-defined interface:

```
class Encoded {
    int value;
    int bitvalue;
    int encoded;
}
```

The variable `value` has to contain the actual value of the syntax element that is encoded with the user-defined datatype. In certain cases, the value of the variable `value` is different with the value that is obtained from the bitstream. Therefore, this value is stored into the `bitvalue` variable. The `encoded` variable can contain two values: 0 or 1. When `encoded` is equal to one, a corresponding Java class has to be created in order to use the resulting bitstream description in the MPEG-21 BSDL framework. Note that this is not necessary when working when generating gBSDs (MPEG-21 gBS Schema).

The following example is the implementation of the unsigned exponential golomb datatype:

```
class UnsignedExpGolomb extends Encoded{
    int leadingzeros = 0;
    while ( nextbits(leadingzeros + 1) == 0 )
        leadingzeros++;
    int length = leadingzeros * 2 + 1;
    int codenum = nextbits(length) - 1;

    bit(length) ue_code;

    int value = codenum;
    int bitvalue = ue_code;
    int encoded = 1;
}
```

1.4 Flow control

BFlavor supports all of the C++ and Java arithmetic, logical, and assignment operators. However, parsable variables cannot be used as lvalues. (Note: this ensures that they always represent the bitstream's contents).

BFlavor also supports all the familiar flow control statements: `if-else`, `do-while`, `while`, and `switch`. In contrast with C++ and Java, variable declarations are not allowed within the arguments of these statements (i.e., `for (int i=0; ; );` is not allowed). The `switch` statement has an extra feature: it is possible to give up a range as value in the `case` statements. This is illustrated in the following example:

```
bit(3) a;
switch(a){
    case 0:
        bit(8) b;
        break;
    case 1 .. 5:
        bit(3) c;
        break;
    default:
        bit(1) d;
```

```
}
```

If the variable `a` has a value in the range of 1 to 5, then a variable `c` of 3 bits is parsed.

Even though Java does not allow non-boolean expressions within the arguments of control statements (e.g., "if (1) ..." is not allowed in Java), BFlavor supports them for less verbose notations. When generating Java code, the translator automatically converts such expressions into corresponding boolean expressions.

The following is an example of the use of these flow control statements.

```
int(2) a;
if (a == 1) {
    int(3) b;
}
else {
    int(4) b;
}
```

The variable `b` is declared with a parse size of 3 if `a` is equal to 1, and with `a` parse size of 4 otherwise.

1.5 Arrays

Multi-dimensional arrays are supported by BFlavor and can be both parsable or non-parsable. The array size does not have to be a constant expression, but it can be a variable as well. The following is thus allowed in BFlavor.

```
int(3) a;
int(2) A[a+1];
```

1.6 Classes

BFlavor uses the notion of classes in exactly the same way as C++ and Java do. It is the fundamental structure in which object data is organized. Keeping in line with the support of both C++ and Java-style programming, classes in BFlavor cannot be nested. Also, due to the declarative nature of BFlavor, methods are not allowed (this includes constructors and destructors).

1.6.1 General

The following is an example of a simple class declaration with just two parsable member variables.

```
class SimpleClass {
    int(3) a;
    int(4) b;
}
```

This class defines objects which contain two parsable variables. They will be present in the bitstream in the same order they are declared. After this class is defined, we can declare objects of this class:

```
SimpleClass s;
```

Class member variables in BFlavor do not require access modifiers (public, protected, private). In essence, all such variables are considered public.

1.6.2 Parameter Types

As BFlavor 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 parenthesis after the name of the class. For example:

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

When declaring variables of parameter type classes, it is required that actual arguments are provided in place of the formal ones:

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

Of course the types of the formal and actual parameters must match.

1.7 Verbatim codes

Verbatim codes are used to signal additional info to the parser that will be generated. Six verbatim codes are defined within the BFlavor specification. These codes are characterized by the verbatim delimiters `%x{` and `%x}`. The symbol `x` hereby denotes the type of the verbatim code.

1.7.1 Namespaces

BFlavor offers two verbatim codes to signal the information about namespaces that have to be used in the resulting description. They have to occur at the beginning of the BFlavor code. Their use is illustrated below.

```
//The namespace used in the resulting description
%targetns{H_264_AVC%targetns}

//The prefix namespace used in the resulting description
%ns{jvt%ns}
```

1.7.2 Root class

The root verbatim code is used to indicate which class the generated parser has to start with. This verbatim code must also be present at the start of the BFlavor code.

```
//The name of the root class
%root{Byte_stream%root}
```

1.7.3 Context class

When a class contains information that is needed for the continuation of the parsing process, we have to declare that this class must be hold in memory (context):

```
%context{index_element%context}
class Example{
   bit(5) index_element;
   bit(2) a;
```

```
   bit(3) b;
}
```

In the above example, we see that a context class is preceded by the `context`-tag. The value of this tag is the name of a variable of the class. This variable is the index of the class. A context class can be uniquely identified with this index element. When a class does not contain an index element (and the active context class is thus always the last occurrence), '0' is given as index element.

1.7.4 Emulation prevention bytes

When a start code occurs coincidentally in the bitstream, emulation prevention bytes can be inserted in order to prevent a start code emulation. BFlavor provides support for emulation prevention bytes by introducing this two verbatim codes. The emulationBytes verbatim code is used by BFlavor to ignore the emulation prevention bytes. A list of couples *(code with emulation prevention byte, code without emulation prevention byte)* is given as an argument to this verbatim code.

```
//indicates that the generated parser has to discard the last
//byte when the byte sequence 0x000301 occurs
%emulationBytes{(000301, 0001);%emulationBytes}
```

The emulateBytes verbatim code is used to signal the occurrence of emulation prevention bytes in the resulted BSD. The argument of this verbatim code is a list of couples *(code which cannot appear in the bitstream, emulated version of the code).*

```
//When the byte sequence 0x0001 has to be written to the bitstream,
//an additional emulation prevention byte (0x03) has to be added.
%emulateBytes{(0001, 000301);%emulateBytes}
```

Note that the verbatim codes dealing with emulation prevention bytes and context information are optional.

1.8 Commenting

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

Chapter 2

gBFlavor specification

So far, we only discussed the BFlavor specification. With this specification, it is already possible to generate a parser that is able to generate gBSDs. More specifically, the resulting gBSD will have the same granularity as the granularity of the gBFlavor code. However, MPEG-21 gBS Schema provides the possibility to add semantically meaningful information to gBSDs by means of markers. In order to target specific applications (i.e., adding markers on well-defined places in the gBSD), the BFlavor specification is extended resulting in the gBFlavor specification.

2.1 The `gBSDApp` environment

The code for the insertion of markers targeting a specific application is denoted by using the keyword `gBSDApp`. For each target application, a list of classes can be specified where a number of calculations have to occur in order to be able to set a marker at the right place. Each class within a `gBSDApp` environment has also to be present in the first part of the gBFlavor code (i.e., the BFlavor code). The code of a particular class occurring in a `gBSDApp` can be considered as a supplement on the code of the corresponding class in the first part of the gBFlavor code. Within each class inside the application-specific environment, a lightweight version of the BFlavor specification can be used together with a new built-in function. Therefore, the following restrictions on the BFlavor specification are applied withing the `gBSDApp` environment:

- parsable variables;
- the built-in functions *skipbits()* and *align()*;

- classes cannot extend the *Encoded* base class;
- classes cannot have parameters
- no verbatim codes are allowed.

Note that most of the restrictions are related to the fact that it is not allowed to parse anything from the bitstream during the assignment of the markers. The following example illustrates the use of gBSDApp environment:

```
// regular BFlavor code containing
// a list of classes

class class1 {
  //...
}

class class2 {
  //...
}


//start of a gBSDApp environment:

gBSDApp ExampleApplication {

  class _class1 {
    //...
  }
}
```

Note that the name of the class within the `gBSDApp` environment is prefixed by the "_" character.

2.2 setmarker() function

Within the `gBSDApp` environment, an additional built-in function is defined. With the `setmarker()` function, it is possible to set a marker at a specific place in the

resulting gBSD. Its use is illustrated in the following example:

```
setmarker("class2", "", "marked_element");
```

This built-in function takes three parameters as input. The first parameter is the name of the class where the marker has to occur. The second parameter is the name of the variable where the marker has to be inserted. In case the second parameter is empty, the marker is inserted at the place where the class starts. In practice this is considered as adding a marker to a *Parameter* element (in case a variable is given in the second parameter) or to a *gBSDUnit* element which corresponds to the given class specified in the first parameter (in case the second parameter is empty). The third parameter is the actual value that will occur in the marker.