BFlavor: Manual

1 Using BFlavor to generate a format specific parser that is able to produce BSDs

1.1 A basic BFlavor code

A first simple example consists of the automatic generation of a parser that is able to describe a text (ASCII) file. The result will be a Bitstream Syntax Description (BSD) containing the different characters occuring in the text file.

Let's first define a description for a text (ASCII) file using BFlavor. Such a file is just a sequence of 8-bit characters. As such, the following BFlavor code describes the syntax of a text file.

```
%targetns{TEXT%targetns}
%ns{text%ns}
%root{CharacterStream%root}

class CharacterStream{
  while(1)
    char(8) character;
}
```

This declares a class called `CharacterStream`, containing an indefinite `while`-loop. In each iteration, a single variable `character` is parsed. The datatype of this variable is `char`, and is represented in the bitstream using 8 bits. Note also the occurrence of the three mandatory verbatim codes at the beginning of the BFlavor code.

Save this in a file called `TextFile.bfl` and run the translator on it by typing: `gbflavorc -gj TextFile.bfl`. The output of the translator will be a Java class stored in the file `CharacterStream.java`. Furthermore, the translator will create an additional class, `Main.java`. This class uses the information from the `root` verbatim code to start the parsing process properly. The class `CharacterStream` will have just one member variable (`character`) and one method: `putxml()`. This method is responsible for translating a bitstream into a corresponding BSD.

The following code shows an excerpt of the Java class that is generated by the gbflavorc translator:

```
public class HelloBits {
```

```
  char character;

  public int putxml(IBitstream _F_bs, Context c) throws IOException {
    ...
  }
}
```

Every class in the BFlavor code is thus translated into a Java class containing a `putxml()` method. This method enables the creation of an XML description for the class member variables.

After compilation of the Java files, we are able to parse a text file. Note that the generated parser always expects two arguments: the input bitstream and the name of the output BSD. Note that in order to properly compile the Java classes, the classpath has to contain the library `flavor.jar`:

```
javac -classpath "PathToFlavorJar/flavor.jar" *.java
```

To run the generated parser, make sure the class files are in the right directory (to avoid problems with packages). The following approach is recommended (in a Win32 environment):

```
gbflavorc.exe -gj flavor_code.bfl
mkdir flavor_code
move *.java flavor_code
javac -cp ".;flavor.jar" flavor_code\*.java
java -cp ".;flavor.jar" flavor_code.Main input.bit output.xml
```

The result of the parsing process for a text file containing the text "HELLO" will result in the following BSD:

```
<CharacterStream
  xmlns="TEXT"
  bs1:bitstreamURI="test.txt"
  xmlns:bs1="urn:mpeg:mpeg21:2003:01-DIA-BSDL1-NS"
  xsi:schemaLocation="TEXT TextFile.xsd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:text="TEXT"
>
```

```
  <character>72</character>
  <character>69</character>
  <character>76</character>
  <character>76</character>
  <character>79</character>
</CharacterStream>
```

Note that this BSD can easily be converted into a bitstream by using the MPEG-21 BSDL software. However, a Bitstream Syntax Schema (BS Schema) is needed. This can be obtained by using the gbflavorc translator:

```
gbflavorc.exe -gx flavor_code.bfl
```

The result is a BS Schema (`flavor_code.xsd`) that can be used by MPEG-21's BSDtoBin parser.

1.2 A more complicated example

We will now continue with a more complicated example which will demonstrate the most constructions that are available in BFlavor. We will build a BFlavor code containing the high-level structure of a fictive video coding format.

1.2.1 The basic structure

```
%targetns{Example%targetns}
%ns{ex%ns}
%root{Byte_stream%root}

class Parameter_Set{
  // ...
}

class Picture (int type){
  // ...
}

class Parse_unit {
  hexBinary(2) start_code = 0x0001;
```

```
  bit(2) type;
  switch(type){
    case 0:
      Parameter_Set parameter_set;
      break;
    case 1 .. 3:
      Picture picture (type);
      break;
  }
}

class Byte_stream{
  while(1){
    Parse_unit parse_unit;
  }
}
```

Our bitstream consists of a collection of parse_units. Each parse_unit contains a start code and a type. Based on this type, the right class is used to continue with the parsing process. The use of `while`, `if-else`, `switch-case` and `for` statements is similar to Java or C++. One extra feature of BFlavor for the `switch-case` construction is the use of ranges in the argument of the `case` statement. The use of the `hexBinary` built-in datatype is demonstrated at the beginning of the class `Parse_unit`. Here, the variable `start_code` is assigned with the value `0x0001`. This means that, during the parsing process, the value of this variable will be checked against the actual value of this variable in the bitstream. Finally, the use of class parameters is demonstrated with the class `Picture` which takes as input the parameter `type`.

1.2.2 User-defined datatypes

```
%targetns{Example%targetns}
%ns{ex%ns}
%root{Byte_stream%root}

%emulationBytes{(000301, 0001);%emulationBytes}
%emulateBytes{(0001, 000301);%emulateBytes}
```

```
class VLC_datatype extends Encoded{
  int temp;
  if(nextbits(2) == 3){
    temp = 1;
  }
  else {
    temp = nextbits(2);
  }

  //implementing the Encoded class:
  int bitvalue = nextbits(temp * 2);
  int value = bitvalue + 2;
  int encoded = 1;

  skipbits(temp * 2);
}

class Parameter_Set{
  VLC_datatype vlc_e1;
  if(vlc_e1.value == 2){
    //...
  }
  align();
}

class Picture (int type){
  if(type == 1){
    //...

  align();
  byteRange(2) payload = 0x0001;
}

class Parse_unit {
  hexBinary(2) start_code = 0x0001;
  bit(2) type;
```

```
  switch(type){
    case 0:
      Parameter_Set parameter_set;
      break;
    case 1 .. 3:
      Picture picture (type);
      break;
  }
}

class Byte_stream{
  while(1){
    Parse_unit parse_unit;
  }
}
```

We extend our BFlavor code by introducing some new constructions that are possible in the BFlavor specification. The most important change is the use of a user-defined datatype: `VLC_datatype`. A user-defined datatype is a class which extends the base class `Encoding`. This class contains three variables: `value`, `bitvalue` and `encoding`. `value` has to contain the value of the variable encoded with the datatype `VLC_datatype`. The variable `bitvalue` contains the bitstream value of the parsed bits. `encoding` can only contain two values: '0' and '1'. If `encoding` is equal to one, then this class needs an additional Java class compatible with the `bs0:implementation` attribute of MPEG-21 BSDL. More information on this topic can be found in the MPEG-21 DIA specification.

Within the `Parameter_Set` class, the use of the `VLC_datatype` datatype is illustrated. Its value is obtained by accessing the `value` variable of the element `vlc_e1`.

There are also some other new constructions introduced here:

- *align()*: provides the possibility to keep the current position in the bitstream byte-aligned. Its use is illustrated at the end of the `Parameter_Set` class. As such, after parsing this class, the current position in the bitstream will be on a byte-aligned position.
- *byteRange*: this built-in datatype is used to discard information we do not want to describe. Using this datatype implies that a pointer to a particular

bitstream segment is added to the BSD. In the class `Picture`, the `byteRange` datatype is used to skip bits in the bitstream until we reach the code '0x0001'. Note that before using a variable of the type byteRange, the bitstream has to be byte-aligned. This can be achieved by using the `align()` built-in function.

- *nextbits(), skipbits()*: within the class `VLC_datatype`, the use of the `nextbits()` and `skipbits()` function is illustrated. Note that the use of `skipbits()` implies that these bits are actually skipped, nothing is written to the BSD (no pointer to a bitstream segment like using the byteRange datatype).

- *emulation prevention bytes*: suppose that the start code '0x0001' can be emulated within the bitstream. In this case, our fictive video coding format can provide the use of emulation preventation bytes. This means that when a start code occurs by accident in the bitstream, one ore more additional bytes are inserted in order to deny the occurrence of start codes in the bitstream (where they not supposed to be). In our example, the start code '0x0001' is converted to '0x000301'. This is signalled by using the `emulationBytes` verbatim code. The second verbatim code, `emulateBytes`, is used to signal the information about the emulation prevention bytes to MPEG-21's BSDtoBin parser.

1.2.3 The use of Context classes

The last extension to our BFlavor code is the introduction of context classes. Consider the BFlavor code for our fictive video coding format:

```
%targetns{Example%targetns}
%ns{ex%ns}
%root{Byte_stream%root}

%emulationBytes{(000301, 0001);%emulationBytes}
%emulateBytes{(0001, 000301);%emulateBytes}

class VLC_datatype extends Encoded{
  int temp;
  if(nextbits(2) == 3){
    temp = 1;
  }
  else {
```

```
    temp = nextbits(2);
  }

  //implementing the Encoded class:
  int bitvalue = nextbits(temp * 2);
  int value = bitvalue + 2;
  int encoded = 1;

  skipbits(temp * 2);
}

%context{class_id%context}
class Parameter_Set{
  bit(8) class_id;
  VLC_datatype vlc_el;
  align();
}

class Picture (int type){
  bit(8) class_id_ref;
  bit( getcontext("Parameter_Set", class_id_ref, vlc_el) ) e1;
  if(type == 1){
    //...

  align();
  byteRange(2) payload = 0x0001;
}

class Parse_unit {
  hexBinary(2) start_code = 0x0001;
  bit(2) type;
  switch(type){
    case 0:
      Parameter_Set parameter_set;
      break;
    case 1 .. 3:
      Picture picture (type);
```

```
      break;
  }
}

class Byte_stream{
  while(1){
    Parse_unit parse_unit;
  }
}
```

In our example, we marked the class `Parameter_Set` as a context class. The reason is that we need information from the `Parameter_Set` class in order to correctly parse the `Picture` class. Since a parse_unit can be either a `Parameter_set` or a `Picture` class, it was not possible to pass this information by using class parameters.

In this fictive video coding format, every picture refers to a specific instance of the class `Parameter_Set` by means of the `class_id_ref` variable occurring in the `Picture` class. This variable corresponds to the variable `class_id` of the class `Parameter_Set`. Note that the variable `class_id` is used as index element in the context class which is denoted by given this variable as argument to the `context` verbatim code.

The retrieval of variables occurring in a context class can be realized by using the getcontext() built-in function as illustrated in the `Picture` class. In this example, the variable `vlc_el` is retrieved from the context.

2 Using BFlavor to generate a format specific parser that is able to produce gBSDs

In order to demonstrate the generation of gBSDs, we reuse the example of the ASCII text file description. Consider the following BFlavor code:

```
%targetns{TEXT%targetns}
%ns{text%ns}
%root{CharacterStream%root}

class ASCII_Character {
  char(8) character;
```

```
}

class CharacterStream{
  while(1)
    ASCII_Character ascii_character;
}
```

This BFlavor code describes the structure of an ASCII text file. With this code, it is already possible to generate a parser that is able to produce gBSDs given a text file. Save this code to `TextFile.bfl` and execute the following commands:

```
gbflavorc -mode 1 -gj TextFile.bfl
mkdir TextFile
move *.java TextFile > NUL
javac -cp flavor.jar TextFile/*.java
java -cp .;flavor.jar TextFile.Main test.txt test.xml
```

We use the option `-mode 1` to indicate that the parser has to produce gBSDs. When we parse a textfile containing the text "HELLO", the following gBSD is generated:

```
<dia:DIA xmlns:dia="urn:mpeg:mpeg21:2003:01-DIA-NS"
         xmlns="urn:mpeg:mpeg21:2003:01-DIA-gBSD-NS">
  <dia:DescriptionMetadata>
    <dia:ClassificationSchemeAlias alias="text" href="TEXT"/>
  </dia:DescriptionMetadata>
  <dia:Description
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:bs1="urn:mpeg:mpeg21:2003:01-DIA-BSDL1-NS"
    xsi:type="gBSDType"
    addressUnit="bit"
    addressMode="Absolute"
    bs1:bitstreamURI="test.txt">
    <gBSDUnit syntacticalLabel=":text:ascii_character" start="0">
      <Parameter name=":text:character" start="0" length="8">
        <Value xsi:type="b8">72</Value>
      </Parameter>
    </gBSDUnit>
    <gBSDUnit syntacticalLabel=":text:ascii_character" start="8">
```

```
      <Parameter name=":text:character" start="8" length="8">
        <Value xsi:type="b8">69</Value>
      </Parameter>
    </gBSDUnit>
    <gBSDUnit syntacticalLabel=":text:ascii_character" start="16">
      <Parameter name=":text:character" start="16" length="8">
        <Value xsi:type="b8">76</Value>
      </Parameter>
    </gBSDUnit>
    <gBSDUnit syntacticalLabel=":text:ascii_character" start="24">
      <Parameter name=":text:character" start="24" length="8">
        <Value xsi:type="b8">76</Value>
      </Parameter>
    </gBSDUnit>
    <gBSDUnit syntacticalLabel=":text:ascii_character" start="32">
      <Parameter name=":text:character" start="32" length="8">
        <Value xsi:type="b8">79</Value>
      </Parameter>
    </gBSDUnit>
  </dia:Description>
</dia:DIA>
```

Note that no markers are occuring in the generated gBSD. In order to make this gBSD more application-specific and therefore easier to transform, we will add a `gBSDApp` environment to the BFlavor code (resulting in a gBFlavor code). This is illustrated in the following example:

```
%targetns{TEXT%targetns}
%ns{text%ns}
%root{CharacterStream%root}

class ASCII_Character {
  char(8) character;
}

class CharacterStream{
  while(1)
    ASCII_Character ascii_character;
```

```
}


gBSDApp CharacterMarking {

  class _ASCII_Character {

    char mychar = $character;
    switch(mychar){
      case 'H':
        setmarker("ASCII_Character", "", "H");
        break;
      case 'E':
        setmarker("ASCII_Character", "", "E");
        break;
      case 'L':
        setmarker("ASCII_Character", "", "L");
        break;
      case 'O':
        setmarker("ASCII_Character", "", "O");
        break;
    }
  }
}
```

The application "CharacterMarking" will put a marker for every ASCII character if the character is equal to 'H', 'E', 'L', or 'O'. The marker will contain the actual character. Note that it is possible to use member variables from the `ASCII_Character` class within the `gBSDApp` environment. However, in that case, such a variable must be prefixed with a `$`. Note also that the class `ASCII_Character` must be prefixed with a _ within the `gBSDApp` environment.

Declaring an application within the gBFlavor code will result in an extra parameter for the generated parser. It is now possible to specify the name of the target application. This name is the name of the corresponding `gBSDApp` environment. The following commands are thus executed:

```
gbflavorc -mode 1 -gj TextFile.bfl
```

```
mkdir TextFile
move *.java TextFile > NUL
javac -cp flavor.jar TextFile/*.java
java -cp .;flavor.jar TextFile.Main
                      test.txt test.xml CharacterMarking
```

By specifying that the target application equals to "CharacterMarking", the following gBSD is generated:

```
<dia:DIA xmlns:dia="urn:mpeg:mpeg21:2003:01-DIA-NS"
         xmlns="urn:mpeg:mpeg21:2003:01-DIA-gBSD-NS">
  <dia:DescriptionMetadata>
    <dia:ClassificationSchemeAlias alias="text" href="TEXT"/>
  </dia:DescriptionMetadata>
  <dia:Description
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:bs1="urn:mpeg:mpeg21:2003:01-DIA-BSDL1-NS"
    xsi:type="gBSDType"
    addressUnit="bit"
    addressMode="Absolute"
    bs1:bitstreamURI="test.txt">
    <gBSDUnit syntacticalLabel=":text:ascii_character" marker="H"
                                              start="0" length="8"/>
    <gBSDUnit syntacticalLabel=":text:ascii_character" marker="E"
                                              start="8" length="8"/>
    <gBSDUnit syntacticalLabel=":text:ascii_character" marker="L"
                                             start="16" length="8"/>
    <gBSDUnit syntacticalLabel=":text:ascii_character" marker="L"
                                             start="24" length="8"/>
    <gBSDUnit syntacticalLabel=":text:ascii_character" marker="O"
                                             start="32" length="8"/>
  </dia:Description>
</dia:DIA>
```

As one can see, targeting a specific application not only results in the occurences of markers within the gBSD. The granularity has also changed resulting in more compact gBSDs. Note that this granularity is calculated based on the places where the markers are set.

3 Command Line Options

The BFlavor translator accepts a number of command line options. In the following we describe its options in detail, and explain how they affect the generated code. Options are grouped in terms of their functional behaviors.

3.1 Usage Message

```
-h, -?  Print the usage message.
```

Prints the usage summary, including the list of the available command line options. The translator only works on one .bfl file at a time and all options must be specified before the input file name. For example:

```
gbflavorc -gj input.bfl
```

The exit value of the translator (the value returned to the operating system) is the number of errors encountered during parsing and code generation. A value of 0 indicates successful operation.

3.2 Version Information

```
-V  Print the version information.
```

Prints a message detailing the current version number and date of release of the translator. For example:

```
GBFLAVORC Version 4.0.0 [beta] 01/11/2006
Copyright (C) 2004-2006, Multimedia Lab, Ghent University.
http://multimedialab.elis.ugent.be
```

3.3 Technology options

```
-gj  Generate Java source classes.
-gx  Generate MPEG-21 BS schema.
```

The `gj` option produces Java classes, i.e., the translator generates a .java file for every BFlavor class defined in the .bfl file. When using the `gx` option, the translator produces an MPEG-21 BS schema for a given BFlavor code. For every class defined in BFlavor, a corresponding complex type is defined.

```
-p  int    set profile (BSDL-1: int=1 (default), BSDL-2: int=2)
-bv int    set BSDL version (BSDL_1.1.3: int=0,
                             BSDL_1.2.1: int=1 (default))
-mode int  set Program mode (BSDL: int = 0 (default),
                             gBS Schema: int = 1)
```

It is possible to generate a BS Schema that cannot only be used by MPEG-21's BSDtoBin parser, but also by MPEG-21's BintoBSD parser. Therefore, the BS Schema needs support for BSDL-2. This option can be enabled by setting the `-p` option on two. One can also choose between different versions of the MPEG-21 BSDL reference software by using the `-bv` option. In order to choose between generating a parser that generates BSDs or gBSDs, the option `-mode` can be used.

3.4 Output File Selection

```
-oj name  Set the output Java package name
-op name  Set the name of the Flavor run-time
          library package (default: flavor).
-ox name  Set the output file name for the generated
          BSDL Schema file.
```

In the case of producing Java code, the translator produces one .java file per BFlavor class and they are packaged using the input file name, by default. Using the -oj name option, the desired package name can be set. The generated Java code utilizes the BFlavor library by importing the library package. By default, the package name is "flavor"; however, this can be modified by using the `-op` option.

Additionally, when generating an MPEG-21 BS schema, the default output file name is the input file name with the suffix changed to .xsd.

3.5 Array Sizing

```
-a size  Set maximum array size. The default is 64.
```

Although BFlavor supports arrays of arbitrary sizes, the translator produces code that declares all arrays with a constant, statically defined size. This option sets the initial value for this size, with the default value of 64. The translator will automatically increase it (and issue a warning message) when it detects that a larger value is needed. Note that this is not always possible, so that care should be taken by the user to ensure that a large-enough size is used.

3.6 Null-Terminated Strings

```
-s  Use null-terminated strings
```

The translator converts strings to arrays, with or without a trailing '\0' (null). This option causes to include the trailing null.

3.7 Run-Time Interface

```
-B string  Use string as the bitstream I/O class.
            The default is IBitstream.
-F string  Use string as a prefix for internal Flavor-generated
            variables. The default is _F_.
-E string  Use function string to report bitstream syntax errors.
            The default is flerror.
```

The generated code relies on a small number of classes and functions to perform low-level bitstream I/O and error reporting. These are part of the run-time library which is included in this package. The design of the run-time library has been done in a way that facilitates its substitution by alernate implementations that may better suit a programmer's application needs. These options allow the customization of the code produced by the translator so that the programmer-supplied substitute classes or functions are called. Note that, although the names can be modified, the interface exposed by these classes must comply to the one expected by `gbflavorc`.

The `-B` option sets the name of the class implementing low-level I/O. The default class, provided in the run-time library, is Bitstream. The `-E` option is used to set the name for the function to be called when bitstream syntax errors occur. The default is `flerror`.

BFlavor also uses a small number of internal variables. To avoid conflicts with variables defined in the user's code, all such variables are given a prefix. The default is `_F_`, but it can be changed using the `-F` option.

3.8 Disallow Further Options

```
--  Disallow further options
```

This options is used to disallow usage of further options. This allows to use a BFlavor source file name that starts with a dash.