VALIDATION OF BITSTREAM SYNTAX AND SYNTHESIS OF PARSERS IN THE MPEG RECONFIGURABLE VIDEO CODING FRAMEWORK

Mickaël Raulet, Jonathan Piat

Institut d'Electronique et de
Télécommunications de Rennes (IETR)
UMR CNRS 6164 (France)
Email: mraulet@insa-rennes.fr

Christophe Lucarz, Marco Mattavelli

Microelectronic Systems Laboratory (GR-LSM)
Ecole Polytechnique Fédérale
de Lausanne (Switzerland)
{christophe.lucarz,marco.mattavelli}@epfl.ch

ABSTRACT

Video coding technology has evolved in the past years into a variety of different and complex algorithms. So far the specification of such standard algorithms has been done case by case providing monolithic textual and reference SW specifications, but without any attention on commonalities and the possibility of incremental improvements or modifications of such monolithic standards. The MPEG Reconfigurable Video Coding (RVC) framework is a new ISO standard, currently under development aiming at providing video codec specifications at the level of library functions instead of monolithic algorithms. The possibility to select a subset of standard coding algorithms to specify a decoder that satisfies application specific constraints is very attractive. However, such possibility to reconfigure codecs requires systematic procedures and tools capable of describing the new bitstream syntaxes of such new codecs. Moreover, it is also necessary to generate the associated parsers which are capable to parse the new bitstreams because they are not available "a priori" in the RVC library. This paper further explains the problem and describes the technologies used to describe new bitstream syntaxes within RVC. In addition, the paper describes the methodology and the tools for the validation of bitstream syntaxes descriptions as well as an example of systematic procedures for the direct synthesis of parsers in the same data flow formalism in which the RVC library component are implemented.

1. INTRODUCTION

Video coding has changed a lot since its infancy in the early nineties. The first original MPEG video coding standard was released in 1993, and since then MPEG-2, MPEG-4 and AVC (Advanced Video Coding) have been produced and SVC (Scalable Video Coding) has been recently standardized. Each successive codec released by MPEG has been substantially more complex than the last, typically yielding twice the compression performance of its predecessor. Because of this growing complexity, the textual specification of recent standards (since MPEG-4) has lost its normative role, being replaced by the reference software implementation as the true normative specification. However, while this normative specification (typically in generic C or C++) is very precise, it presents a number of limitations. Large portions of compression technology (i.e. coding tools) are common across all MPEG standards, yet there is no direct way to recognize or exploit this commonality. Additionally, the sequential C/C++ descriptions do not expose the potential parallelism that is intrinsic to the algorithms constituting the codecs. They have also become excessively large (in terms of code size), making it extremely time consuming to transform the sequential reference software into a VHDL implementation or to map it onto a multicore platform. In other words, the complex sequential C/C++ specifications no longer constitute a good starting point for the implementation processes of standard video codecs on current and future platforms. The challenge taken by the Reconfigurable Video Coding (RVC) framework currently under development by MPEG is to provide a high level specification model for direct and efficient software and hardware synthesis.

The essential elements of the RVC framework are the following:

- A library of video coding tools, also called Functional Units (FUs) covering all MPEG standards (the "MPEG Toolbox"). This library is specified and provided using CAL as specification language for each library component (i.e. video coding tool) [1, 2] CAL [1] is a language used to define the behavior of dataflow components called actors, which is a modular component that encapsulates its own state such that an actor can neither read nor modify the state of any other actor. The only interaction between actors is via messages (known in CAL as tokens) which are passed from an output of one actor to an input of another. The behavior of an actor is defined in terms of a set of actions, transactional program fragments, at most one of which may be active at any point in time inside an actor. The operations an action can perform are to consume (read) input tokens, modify internal state, produce output tokens, and inter-

978-1-4244-2924-0/08/$25.00 ©2008 IEEE

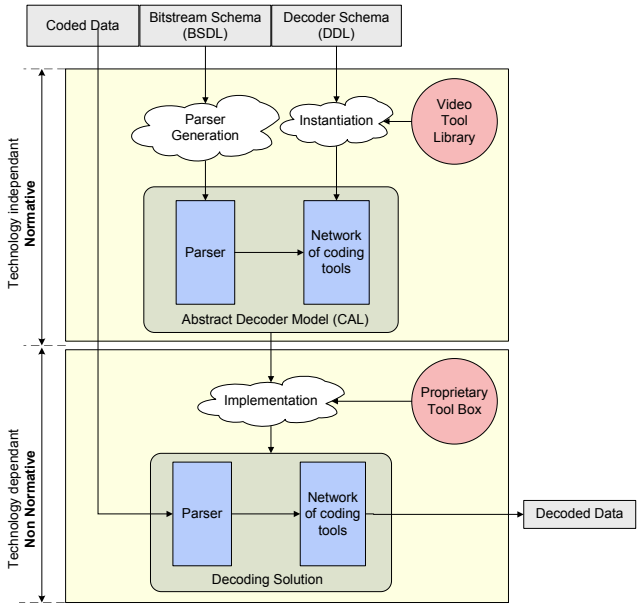

Fig. 1. The Reconfigurable Video Coding framework

act with the underlying platform on which the actor is running.

- A language that provides the description of video codec representations called Decoder Description Language (DDL). This is an XML dialect that describes an interconnected network and parameterization of standard library components, which together represent a complete decoder. DDL can also be used recursively; that is, an actor may be defined as a composition of other actors, with the interconnections specified by DDL. In this case, the DDL itself declares input and output ports. DDL provides a facility for declaring parameters, and passing parameters to actors in the network. This is useful for declaring values that are constant for a particular instantiation of an actor, but may vary between different instantiations. An "abstract model" is constituted by the instantiation of a codec configuration using the Decoder Description Language and the MPEG Toolbox. Figure 1 depicts the process of instantiating an "abstract decoder model" in RVC.

- Tools capable to verify and validate the behavior of the "abstract model" and tools capable to generate automatically software and hardware descriptions of the abstract model.

An important problem faced by RVC is how to describe and specify a new bitstream syntax and how to generate the associated parser in CAL. In fact all components of any codec reconfiguration can be found in the RVC toolbox except the parser. Without systematic procedures and support tools for the validation of new bitstream syntaxes and the possibly automatic generation of parsers, RVC framework would lack the appropriate elements for a successful usage and deployment.

The paper is organized as follows: section 2 describes the essential elements of BSDL (a MPEG-21 standard language used to specify a new bitstream syntax). Section 3 describes a procedure for the validation of BSDL schemas. Section 4 reports how it is possible to automatically generate a parser in a form compatible with the RVC ADM from a BSDL schema by using a XSLT transformation. Section 5 concludes the paper.

2. BSDL A LANGUAGE TO DEFINE BITSTREAM SYNTAX

MPEG-B part 5 is an ISO/IEC international standard that specifies BSDL [3] (Bitstream Syntax Description Language), an XML Schema describing the generic bitstream syntax, which can then be used to extract an XML implementation from a binary bitstream. For instance, in the case of a MPEG-4 AVC video codec [4], a BS Schema describes the structure common to all possible conformant MPEG-4 AVC video bitstreams, whereas a BS description describes a single MPEG-4 AVC encoded bitstream as a XML document. Figure 2(a) shows the BSDL Schema associated with the BSDL Description in Figure 2(b). BSDL uses XML to describe the structure of video coded data. An encoded video bitstream can be described as a sequence of binary symbols of arbitrary length – some symbols contain a single bit, while others contain many bytes. For these binary symbols, the BSDL Description indicates values in a human – and machine – readable format – for example, using hexadecimal values (as for startCode in Figure 2(a)), integers, or strings. It also organizes the symbols into a hierarchical structure that reflects the data hierarchic/semantic interpretation.

In other words, the BSDL Description level of granularity can be fully customized to the application requirements [5]. BSDL was originally conceived and designed to enable adaptation of scalable multimedia content in a format-independent manner [6]. In the RVC framework, BSDL is used to fully describe the entire bitstream – each elementary bit has its corresponding value in a Variable Length Decoding (VLD) table. As a result, the corresponding BS schema must specify all components of the syntax at a finer granularity level than the ones developed and used for adaptation of scalable content. In this context BSDL does not replace the original data, but instead provides additional information (or metadata) to support an application for parsing and processing the binary content. Finally, BSDL does not mandate the names of the elements in the BSDL Description; the application assigns names that provide meaningful semantics for the description at hand. Figure 2(a) is an example BSDL Description for video in MPEG-4 AVC format.

In the RVC framework, BSDL is preferred over Flavor [7]

because:

- it is stable and already defined by an international standard;
- the XML-based syntax integrates well with the XML syntax used to describe the configuration of the RVC decoder; constituted by the instantiation of FUs from the toolbox and by their connettivity;
- the RVC bitstream parser may be easily derived by transforming the BSDL schema using standard tools (e.g. XSLT).

```
<NALUnit>
  <startCode>00000001</startCode>
  <forbidden0bit>0</forbidden0bit>
  <nalReference>3</nalReference>
  <nalUnitType>20</nalUnitType>
  <payload>5 100</payload>
</NALUnit>
<NALUnit>
<startCode>00000001</startCode>
<!-- and so on... -->
</NALUnit>
```

(a) BS description fragment of an MPEG-4 AVC bitstream

```
<element name="NALUnit"
  bs2:ifNext="00000001">
<xsd:sequence>
  <xsd:element name="startCode" type="avc:hex4" fixed="00000001"/>
  <xsd:element name="nalUnit" type="avc:NALUnitType"/>
  <xsd:element ref="payload"/>
</xsd:sequence>
<!--  Type of NALUnitType -->
<xsd:complexType name="NALUnitType">
  <xsd:sequence>
    <xsd:element name="forbidden_zero_bit" type="bs1:b1" fixed="0"/>
    <xsd:element name="nal_ref_idc" type="bs1:b2"/>
    <xsd:element name="nal_unit_type" type="bs1:b5"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:element name="payload" type="bs1:byteRange"/>

<!-- and so on... -->
```

(b) BS schema fragment of MPEG-4 AVC codec

Fig. 2. BSDL description and schema

The RVC framework aims at supporting the development of new MPEG standard and new decoding solutions. The flexibility offered by the standard video coding library to explore rapidly the design space is primordial. Defining coding tools and their interconnections becomes a relatively easy task if compared to the SW rewriting efforts need to modify (usually very large) monolithic specifications. However, testing new decoding solutions, new algorithms for new coding tools, or new tools configurations, the bitstream syntax may change from a solution to another. The consequence is that a new parser need to be rewritten for each new bitstream syntax. The parser FU is the most complex actor in the MPEG-4 SP decoder [4] described in [8] and its behavior need to be validated versus all possible conformat bistreams. This is equivalent to validate it using the BSDL schema for the syntax at hand. Moreover, it is certainly not a good idea to have to write it by hand when a systematic solution for deriving such parsing procedure from the BSDL schema itself could be developed. Such procedure based on transforming the BSDL schema by a XSLT transformation is describes in the second part of the paper. So being able to validate a parsing procedure (written by hand or automatically generated) using some instances of a given syntax is an important step for the RVC framework.

3. VALIDATION OF A BSDL SCHEMA BISTREAM SYNTAX DESCRIPTION

3.1. Procedure of validation

The generic character of BSDL, and hence its merit, lies in the media format-independent nature of the different software modules that are responsible for the creation of the BS Descriptions (BSDs) and for the generation of the adapted bitstreams. The BSD generator and bitstream generator are named BintoBSD Parser and BSDtoBin Parser, respectively. Figure 3 summarizes the overall method for validating a BS schema. Explanatory notes for this figure are provided below:

1. a bitstream syntax schema (BS Schema) contains a description of the low-level syntax in RVC of a particular media format;
2. a BSD is created by a format-independent BintoBSD Parser, taking as input a particular bitstream and a corresponding BS Schema;
3. a BSD is transformed to meet the constraints of a usage environment;
4. a format-independent BSDtoBin Parser creates the original bitstream, using the transformed BSD and the BS Schema

There are two possibilities to compare the efficiency of the schema:

1. the original bitstream is compared to the one produced by the identity operation “bintoBSD-BSDtobin”; this bitstream should give the same decoded sequence as the original one;
2. the BSD description generated after the first “bintoBSD” operation could be compared to the identity operation “BSDtobin-bintoBSD”. You should exactly obtain the same BS Descriptions.

A BS Schema contains a minimal amount of information such that BSDtoBin can convert each element value in a BSD to a bit-level representation. Such functionality can already be provided by an XML Schema using BSDL-1 datatypes, as BSDL-2 is specific for BintoBSD and not relevant for BSDtoBin. Thus, BSDtoBin may still be used for generating a bitstream to support BSDL-2.

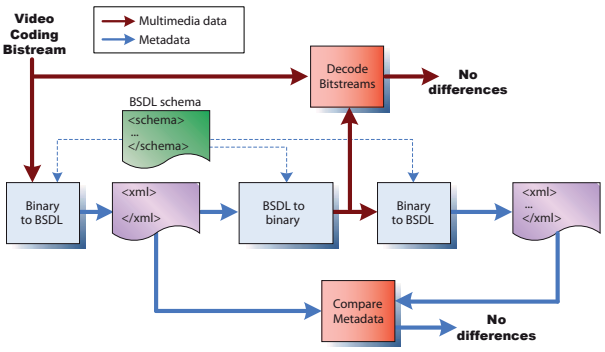

Fig. 3. BS Schema validation

3.2. User-defined data types

This subsection specifies an optional implementation mechanism for user-defined data types, that a conformant BSDtoBin or BintoBSD parser does not have to implement. But if an ECMAScript implementation of bs1:codec data types is provided, the parser shall conform to this clause. Data types referenced by bs1:codec in a BS Schema may be implemented using ECMAScript and the implementation is embedded in the BS Schema via the bs1:script component. This allows arbitrary parsing algorithms to be specified by a BS Schema for use by BintoBSD and BSDtoBin parsers, enabling the processing of data structures that cannot be specified using other BSDL syntax elements. The bs1:script component defines the local name of the datatype, which inherits the target namespace of the schema document. The bs1:codec attribute can then reference this implementation via the URI of the datatype, which is obtained by adding the appending the local name as fragment identifier to the namespace.

ECMAScript datatypes may be used to allow a BSDL Parser to process Variable Length Codes, such as Huffman codes or Arithmetic-coded values (Figure 5). An ECMAScript implementation may be referenced by bs1:codec in the following ways:

- the value of bs1:codec is a URL that resolves to a BS Schema, with a fragment identifier corresponding to the value of an id attribute on a bs1:script element;
- the value of bs1:codec is a URL that resolves to an ECMAScript file, with a fragment identifier corresponding to the name of a class within that file;
- the value of bs1:codec is a URL that resolves to an ECMAScript file, with no fragment identifier.

In each case, a BSDtoBin parser shall search the bs1:script element, class or file (respectively) for a function (or method) with the signature BSDtoBin(value). The BSDtoBin parser shall call this function to parse the element to which bs1:codec is attached. The BSDtoBin parser shall pass as value the text content of the element if the content is simple, or the element and its descendents, otherwise.

A BintoBSD parser shall search the bs1:script element, class or file (respectively) for a function (or method) with the signature BintoBSD(). The parser shall call this function to generate the element to which bs1:codec is attached, the BintoBSD() function should return either a string containing the lexical value of the element, or a DOM Element representing the element.

read(bits) This function shall be provided by a BintoBSD parser and may be called by the BintoBSD() function of a bs1:script component. When this function is called, a BintoBSD shall read from the bitstream the number of bits specified by the integer value of the bits parameter, and return the unsigned integer value of the bits read.

write(value,bits) This function shall be provided by a BSDtoBin parser and may be called by the BSDtoBin(value) function of a bs1:script component.

xpath(exp,type) This function shall be provided by a BintoBSD parser and may be called by the BintoBSD() function of a bs1:script component. When this function is called, a BintoBSD shall execute the XPath expression declared by the string value of the exp parameter, and return the value of the result of the expression. The expression shall be evaluated in the context of the partially instantiated BSD.

```
<xsd:complexType name="expGolomb">
  <xsd:simpleContent>
    <xsd:extension base="xsd:unsignedInt">
      <xsd:attribute ref="bs1:codec" default="expGolomb.js"/>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>
```

(a) Javascript call outside BintoBSD tool

```
<xsd:complexType name="expGolomb">
  <xsd:simpleContent>
    <xsd:extension base="xsd:unsignedInt">
      <xsd:attribute ref="bs1:codec"
          default="urn:mpeg:example:myLibrary#expGolomb"/>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>
```

(b) Java Class call inside BintoBSD tool

Fig. 4. Implementation of expgolomb function

In the case you use ECMAscript implementation in your schema, you will have to validate your script in the two ways: bintoBSD and BSDtobin.

The following implementation of Expglomb does not need but only one ECMAscript file containing 2 parts:

- the ECMAscript function called by bintoBSD (Figure 5(a));

- the reverse ECMAscript function called by BSDtobin (Figure 5(b)).

```
function BintoBSD() {
    var nBits = 0;
    var ret = 0;

    while ((ret = read(1)) == 0) nBits++; //read 0's
    if (ret == -1) throw "userType Error";
    ret = read(nBits);  //read the rest
    if (ret == -1) throw "userType Error";
    return ((1 << nBits) - 1 + ret) + ""; //toString
};
```

(a) ECMAscript implementation of expgolomb for bintoBSD tool

```
function BSDtoBin(value) {
    var nBits = 0;
    var tmp = value + 1;

    while ((tmp >>= 1) > 0) nBits++; //count how many zeros to write
    tmp = 1;
    var i = 0;
    for (i = 0; i < nBits; i++) {
        tmp <<= 1;
    }
    write(0, nBits); //write leading zeros
    write(1, 1); //write a one
    write(value + 1 - tmp, nBits); //write rest of code
    return(2 * nBits + 1);
}
```

(b) ECMAscript implementation of expgolomb for BSDtobin tool

Fig. 5. ECMAscript implementation of expgolomb function

4. SYNTHESIS OF A PARSER IN CAL FROM A BSDL SCHEMA DESCRIPTION

Writing a complete parser by hand is a burdensome, time consuming and error prone task. It is certainly not a smart solution. Once the bitstream is validated (section 3), a systematic methodology generates automatically a parser in CAL from the bitstream schema (in BSDL). This idea has been first presented in [8]. The generated code is now compatible with hardware and software code generators. It was not the case in the last version. Furthermore, variable length decoding is supported and additional BSDL constructs. Figure 7 illustrates the different steps of the transformation process and an example of the result of the generation. The advantage of generating the parser in CAL is that the entire decoder model is thus described in the same formalism. Direct synthesis of the CAL decoder model to SW or HW implementations can be performed [9, 10] from this complete model. The reader can also refer to [8] for a more detailed background and further details on the parser generation process.

Figure 8 shows an example of bitstream schema from which a parser has been generated. The resulting CAL code, generated by the tool, is shown figure 6. This CAL code executes the bitstream parsing.

Each time a syntax element is met by the parser, the process generates a xxxx.read" action. If this element of syntax

```
mcbpc.read: action ⇒
guard
 readDone()
do
 current := read_result_in_progress ;
 setRead(M4V_B1_LENGTH);
end

ac_pred_flag.read: action ⇒
guard
 readDone()
do
 current := read_result_in_progress ;
 setRead(M4V_B2_LENGTH);
end

cbpy.read: action ⇒
guard
 readDone()
do
 current := read_result_in_progress ;
 setRead(M4V_B3_LENGTH);
end

dct_dc_size.read: action ⇒
guard
 readDone()
end

dct_dc_size.output: action ⇒ size: [current]
do
 current := read_result_in_progress ;
 setRead(M4V_B4_LENGTH);
end

dct_dc_diff.read: action ⇒
guard
 readDone()
end

dct_dc_diff.output: action ⇒ diff: [current]
do
 current := read_result_in_progress ;
 setRead(M4V_VLC_LENGTH);
end

DCT_coeff.read: action ⇒
guard
 readDone()
end

DCT_coeff.output: action ⇒ coeff: [current]
do
 current := read_result_in_progress ;
end

DCT_coeff.finish: action coeff_f: [f] ⇒
guard
 f = 2
do
 setRead(M4V_B3_LENGTH);
end

DCT_coeff.notFinished: action coeff_f: [f] ⇒
guard
 f = 0 or f  = 1
do
 setRead(M4V_VLC_LENGTH);
end

// Finite State machine

mcbpc_exists (mcbpc.read) --> ac_pred_flag_exists;
ac_pred_flag_exists (ac_pred_flag.read)--> cbpy_exists;
cbpy_exists     (cbpy.read) --> dct_dc_size_exists;
dct_dc_size_exists (dct_dc_size.read)--> dct_dc_size_output;
dct_dc_size_output (dct_dc_size.output)-->dct_dc_diff_exists;
dct_dc_diff_exists (dct_dc_diff.read)-->  dct_dc_diff_output;
dct_dc_diff_output (dct_dc_diff.output)-->DCT_coeff_exists;
DCT_coeff_exists (DCT_coeff.read)--> DCT_coeff_output;
DCT_coeff_output (DCT_coeff.output)--> DCT_coeff_result;
DCT_coeff_result (DCT_coeff.notFinished)--> DCT_coeff_exists;
DCT_coeff_result (DCT_coeff.finish)--> next_elements;
```

Fig. 6. Source code of the parser generated

Fig. 7. The XSLT transformation process

```
<xsd:complexType name="MB">
  <xsd:sequence>
  <xsd:element name="mcbpc" type="b3"/>
  <xsd:element name="ac_pred_flag" type= "b1"/>
  <xsd:element name="cbpy" type= "b2"/>

  <xsd:complexType name="block">
     <xsd:sequence>
          <xsd:element name="dct_dc_size" type="b3" rvc:port="size"/>
         <xsd:element name="dct_dc_diff" type="b4" rvc:port="diff" />
         <xsd:element name="DCT_coeff" type="vlc" rvc:port="coeff" />
      </xsd:sequence>
  </xsd:complexType>
```

Fig. 8. Example of BSDL description

must be presented at the output by the parser, a xxxx.output" action is created. When the parser meets a variable length code, it creates a series of actions which are necessary to communicate with the VLD FUs: xxxx.read" to read the bit from the input port, xxxx.output" to send the bit the the VLD table, and xxxx.finished" / xxxx.notfinished" to decide if the variable length code is finished of if the parser must send an additional bit. To get more information about the implementation of variable length decoding process in RVC, the reader can refer to the paper [11].

5. CONCLUSION

This paper describes a systematic methodology for the validation of a BSDL schema describing the syntax of a binary bitstream. The validation of a BSDL schema is a fundamental step in the RVC framework. The validated schema is the input of a tool that generates automatically a CAL parser. This latter completes the model used to specify, design, and implement decoders in the RVC framework.

6. REFERENCES

[1] J. Eker and J. Janneck, "CAL Language Report," ERL Technical Memo UCB/ERL M03/48, 2003.

[2] J. W. Janneck, *The CAL actor language: Synthesizing models to FPGA*. http://chess.eecs.berkeley.edu/pubs/181.html.

[3] International Standard ISO/IEC FDIS 23001-5, *MPEG systems technologies - Part 5: Bitstream Syntax Description Language (BSDL).*

[4] *ISO/IEC14496 Coding of audio-visual objects*. 2004.

[5] J. Thomas-Kerr, J. Janneck, M. Mattavelli, I. Burnett, and C. Ritz, "Reconfigurable Media Coding: Self-Describing Multimedia Bistreams," in *IEEE Workshop on Signal processing Systems SiPS 2007*, (Shanghai, China), April 17-19, 2007 2007.

[6] J. Thomas-Kerr and I. Burnett and C. Ritz and S. Devillers and D. De Schijver and R. Van de Walle, "Is That a Fish in Your Ear? A Universal Metalanguage for Multimedia," *IEEE Multimedia*, vol. 14(2), pp. 72–77, 2007.

[7] A. Eleftheriadis, "Flavor: A Language for Media Representation," *ACM Int'l Conf. on Multimedia*, pp. 1–9, 1997.

[8] C. Lucarz, M. Mattavelli, J. Thomas-Kerr, and J. Janneck, "Reconfigurable Media Coding: A New Specification Model for Multimedia Coders," in *IEEE Workshop on Signal Processing Systems*, pp. 481–486, 2007.

[9] J. W. Janneck, I. D. Miller, D. B. Parlour, M. Mattavelli, C. Lucarz, M. Wipliez, M. Raulet, and G. Roquier, "Translating Dataflow Programs to Efficient Hardware: an MPEG-4 Simple Profile Decoder Case Study," in *Design, Automation and Test in Europe (DATE)*, (Munich, Germany), 2008.

[10] M. Wipliez, G. Roquier, M. Raulet, J.-F. Nezan, and O. Déforges, "Code generation for the MPEG reconfigurable video coding framework: from CAL actions to C functions," in *IEEE International Conference on Multimedia & Expo (ICME)*, (Hannover, Germany), 2008.

[11] J. Li, D. Ding, C. Lucarz, S. Keller, and M. Mattavelli, "Efficient Data Flow Variable Length Decoding Implementation For The Mpeg Reconfigurable Video Coding Framework," in *IEEE Workshop on Signal Processing Systems*, (Washington DC, US), 2008.