Author manuscript, published in "Journal of Signal Processing Systems (2009) online"
DOI : 10.1007/s11265-009-0395-7

hal-00407949, version 1 - 28 Jul 2009

**Noname manuscript No.**
(will be inserted by the editor)

# Automatic synthesis of parsers and validation of bitstreams within the MPEG Reconfigurable Video Coding Framework

**Christophe Lucarz** · **Jonathan Piat** · **Marco Mattavelli**

Received: date / Accepted: date

**Abstract** Video coding technology has evolved in the past years into a variety of different and complex algorithms. So far the specifications of such standard algorithms have been done case by case, providing monolithic textual and reference software specifications, but without paying any attention the possibility of further improvements of such monolithic standards. The MPEG Reconfigurable Video Coding (RVC) framework is a new ISO standard, currently under its final stage of development aiming at providing video codec specifications at the level of coding tools instead of monolithic descriptions. The possibility to select a subset of standard video 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 becomes also necessary to generate the associated parsers, capable of parsing the new bitstreams. This paper further explains the problem and describes the technologies used to describe new bitstream syntaxes. Additionally, the paper describes the methodologies and the tools for the validation of bitstream syntaxes descriptions as well as a systematic procedure for automatically synthesizing parsers from the bitstream descriptions.

Christophe Lucarz
Microelectronic Systems Laboratory
École Polytechnique Fédérale de Lausanne, Switzerland
E-mail: christophe.lucarz@epfl.ch

Jonathan Piat
IETR/INSA Rennes
F-35043, Rennes, France
E-mail: Jonathan.Piat@insa-rennes.fr

Marco Mattavelli
Microelectronic Systems Laboratory
École Polytechnique Fédérale de Lausanne, Switzerland
E-mail: marco.mattavelli@epfl.ch

## 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 its final standardization stage by MPEG is to provide a high level specification model for direct and efficient software and hardware synthesis.

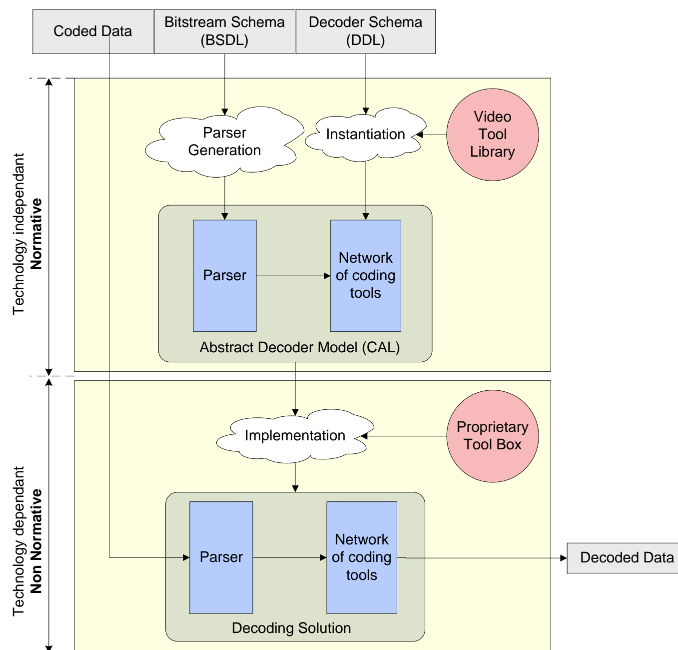


**Fig. 1** The Reconfigurable Video Coding framework

hal-00407949, version 1 - 28 Jul 2009

### 1.1 Essential concepts in RVC

The essential concepts of the RVC framework are the following:

- RVC-CAL [1] , a subset of the CAL data flow language [2] for describing the Functional Unit. This language defines 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 flow from an output of one actor to an input of another. The behavior of an actor is defined in terms of a set of atomic actions. The execution inside an actor is purely sequential: at any point in time, only one action can be active inside an actor. An action can consume (read) tokens, modify the internal state of the actor, produce tokens, and interact with the underlying platform on which the actor is running.
- FNL (Functional Unit Network Language) [1], a language describes the video codec configurations. FNL is a XML dialect that lists the FUs composing the codec, the parametrization of these FUs and the connections between the FUs. FNL allows hierarchical constructions: an FU can be defined as a composition of other FUs and described by another FND (FU Network Description).
- BSDL (Bitstream Syntax Description Language) [3], a language for describing the structure of the input bitstream. BSDL is a XML dialect that lists the sequence of the syntax elements with possible conditioning on the presence of the elements, according to the value of previously decoded elements. BSDL is further explained in section 2.
- A library of video coding tools [4], also called Functional Units (FU) covering all MPEG standards (the “MPEG Toolbox”). This library is specified and provided using RVC-CAL (a subset of the original CAL language) as specification language for each FU.
- An “Abstract Decoder Model” (ADM) is constituted by the instantiation of a codec configuration (described using FNL) 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 Decoder Model (Open DataFlow environment [5]).
- Tools capable to generate automatically software and hardware descriptions of the Abstract Decoder Model

### 1.2 Definition of the problem

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 efforts in rewriting (very large) monolithic software 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 needs to be rewritten for each new bitstream syntax. The parser FU is the most complex actor in the MPEG-4 Simple Profile decoder [6] described in [7] and its behavior needs to be validated with all possible conforming bitstreams. Validating the parser behavior and the BSDL schema by hand become very burdensome tasks. 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 can be developed. Such procedure based on transforming the BSDL schema by a Extensible

hal-00407949, version 1 - 28 Jul 2009

Stylesheet Language (XSL) Transformation is described in the second part of the paper. But being able to validate a bitstream description (written by hand or automatically generated) must be a preliminary step and is described in the first part of the paper.

The paper is organized as follows: section 2 gives an overview of BSDL. 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 Abstract Decoder Model from a BSDL schema by using standard tools (i.e. XSL Transformation). Section 5 concludes the paper.

## 2 BSDL, a language to define bitstream syntaxes

MPEG-B Part 5 is an ISO/IEC international standard that specifies BSDL [3] (Bitstream Syntax Description Language), a XML dialect describing generic bitstream syntaxes. In the field of video coding, the bitstream description in BSDL of MPEG-4 AVC [6] bitstreams represents all the possible structures of the bitstream which conforms to MPEG-4 AVC. A Binary Syntax Description (BSD) is one unique instance of the BSDL description. It represents a single MPEG-4 AVC encoded bitstream: it is no longer a BSDL schema but a XML file showing the data of the bitstream. Figure 2(a) shows a BSD associated to the corresponding BSDL schema shown in Figure 2(b).

```
<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) Bitstream Syntax Description (BSD) 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"/>
```

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

**Fig. 2** A BSD and its corresponding bitstream schema is BSDL

An encoded video bitstream is described as a sequence of binary elements of syntax of different lengths: some elements contain a single bit, while others contain many bits.

hal-00407949, version 1 - 28 Jul 2009

The Bitstream Schema (in BSDL) indicates the length of these binary elements in a human and machine-readable format (hexadecimal, integers, strings ...). For example, hexadecimal values are used for start codes as shown in Figure 2(a). The XML formalism allows organizing the description of the bitstream in a hierarchical structure. The Bitstream Schema (in BSDL) can be specified at different levels of granularity. It can be fully customized to the application requirements [8]. BSDL was originally conceived and designed to enable adaptation of scalable multimedia contents in a format-independent manner [9]. In the RVC framework, BSDL is used to fully describe video bitstreams. Thus, BSDL schemas must specify all the elements of syntax, i.e. at a low level of granularity. Before the use of BSDL in RVC, the existing BSDL descriptions described scalable contents at a high level of granularity. Figure 2(a) is an example BSDL description for video in MPEG-4 AVC format.

The choice of a language to describe the syntax of the bitstream has been firstly discussed in [8]. As a result, BSDL has been preferred over Flavor and XFlavor [10, 11] because:

- it is stable and already defined by an international standard [3];
- in XFlavor, the bitstream is described with a set of classes, in the object-oriented paradigm (C++ or java). The parsing is accomplished by the C++ or Java code generated from the Flavor description. The object-oriented paradigm has not been adopted by the RVC framework. Thus, the XML-based BSDL description of the bitstreams has been chosen because it does not operate a change of paradigm within a same framework.
- the XML-based description of BSDL interacts better with the XML-based representation of the configuration of RVC decoders (FU Network Descriptions) and the XML-based infrastructure of the existing tools. It makes the platform more coherent, with a same technology.

## 3 Validation of a BSDL bitstream schema

Before generating parsers from bitstream schemas, one must guarantee that the schemas are correct, i.e. the schema reflects perfectly the structure of the compressed data which is sent with. If there is no validation procedure, which guarantees that the bitstream is structured exactly as the schema describes it, the generated parser may not be able to parse the bitstream. Thus, a validation procedure is necessary. Figure 3 summarizes the overall method for the validation of bitstreams schemas. Two tools are involved in the validation of such schemas: the BintoBSD parser creates a Binary Syntax Description (BSD) from a particular bitstream and its corresponding Bitstream Schema (BS). The reference decoder implementation outputs the reference BSD, representing the right structure of the data contained in the bitstream. The validation of the BSDL schema consists in comparing the reference BSD (generated by the reference decoder implementation) and the BSD generated by the BintoBSD parser from the Bitstream Schema. If the two BSD are identical, it means that the bitstream follows correctly the schema under evaluation. If this procedure is repeated on a set of input bitstreams which cover the whole BSDL schema, the schema is considered as correct.

### 3.1 The case of unsized elements of syntax

In many cases, the size in bits of a syntax element is not known neither at compile-time nor at run-time (e.g. Variable Length Codes). The size in bits of the syntax element is known

hal-00407949, version 1 - 28 Jul 2009

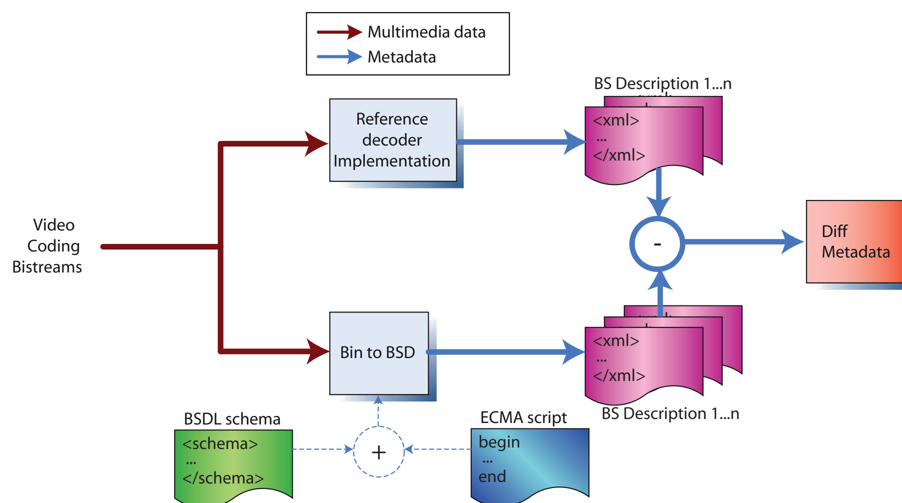


**Fig. 3** Illustration of the Bitstream Schema validation procedure

only during the decoding process of this syntax element. The validation process implies the generation of a BSD from a given bitstream: it means at least the determination of the size of every syntax element. For the elements whose size is known *a priori*, generating the BSD is straightforward. But for the elements whose size is not known *a priori*, a parsing algorithm must be implemented in order to determine the size of the elements of syntax. For the validation process, these parsing algorithms are written in Javascript and are linked to the bitstream schema (in BSDL) by means of the *bs1:codec* and *bs1:script* BSDL constructs. Data types with the attribute *bs1:codec* in a bitstream schema are decoded using ECMAScript and the implementation is embedded in the bitstream schema via the *bs1:script* BSDL construct. It allows parsing algorithms to be specified in a bitstream schema and used by BintoBSD, enabling the processing of data structures that cannot be specified with BSDL constructs.

Figure 4(a) shows an example of declaration of an user-defined element (”expGolomb”) for which a Javascript parsing algorithm is necessary for decoding it. The code of the parsing algorithm in Javascript decoding the “ExpGolomb” syntax element is provided in figure 4(b).

The BintoBSD tool searches the *bs1:script* element, class or file (respectively) for a function (or method) with the signature BintoBSD(). The tool calls this script to generate the element value to which the *bs1:codec* attribute is attached. BintoBSD() function returns a string containing the lexical value of the element and modifies the Xpath variables contained in the BS schema – Xpath is standardized as an extended feature in [3]. The number of bits of the elements is consumed and the process of generation of the BS description can continue. Entropic decoders such as CAVLC and CABAC in MPEG4-AVC requires some contextual values from the already decoded macroblocks. Implementation of BintoBSD provide the ECMAScript with functions allowing to:

– evaluate Xpath expression inside the BS schema,
– evaluate Xpath expression inside the outputted BS description,
– modify Xpath variable values.

hal-00407949, version 1 - 28 Jul 2009

```
<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) Call of a Javascript function inside BSDL

```
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
};
```

(b) Example of a Javascript function: ExpGolomb

**Fig. 4** Call and example of a parsing algorithm in Javascript

So Xpath and ECMAScript provide the way to resolve these contextual values.

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.

For instance, ECMAScript datatypes may be used to allow a BSDL parser to process Variable Length Codes, such as Huffman codes or Arithmetic-coded values (Figure 4(b)). 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 Bitstream 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.

## 4 Synthesis of parsers from a bitstream description

The bitstream structure is described in BSDL, a XML dialect (section 2). The bitstream needs first be validated (section 3). XSL Transformation is a language used for the transformation of XML documents into other XML documents. CAL can be also represented in a XML format: CALML. Thus, XSL Transformations are perfectly adapted to convert a bitstream schema written in BSDL into a parser in CALML. The difficulty of the transformation remains in the fact that a *description* (the schema) is converted into an *executable*: the bitstream schema (in BSDL) describes in the XML formalism the sequence of syntax elements constituting the bitstream. There is no indication on how to parse them. The parser is an executable that processes these elements of syntax. It is very difficult to make the transformations sufficiently flexible such as the resulting executable (the parser) allows all the combinations of the BSDL constructs constituting the schema.

hal-00407949, version 1 - 28 Jul 2009

Furthermore, the generated CALML code can be used as an input to the hardware and software code generators [12, 13]. Thus, direct synthesis of the parser into hardware or software implementations can be performed.

Figure 5 illustrates the different steps of the XSL Transformation process.

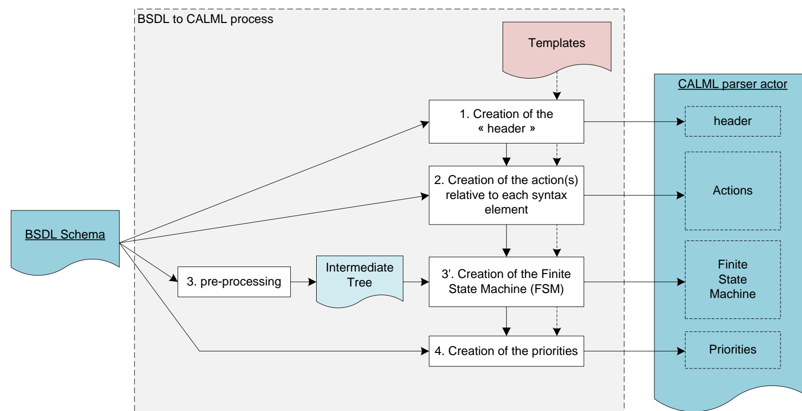


**Fig. 5** Block diagram of the XSL Transformation process.

The BSDL to CALML transformation is composed of four main steps. At each step of the process, the BSDL schema is traversed and only some parts are transformed according to the step. For the step 1, 2, 3' and 4, CALML templates are used to create the final CALML parser. These templates are filled according to each syntax element.

The first step in the transformation is the creation of the header of the parser actor in CALML. It consists of adding constant values, initialized variables, input and output ports and the signature of the actor.

The second step is the creation of the actions for each syntax element of the BSDL schema. One or several actions can be created for each syntax element. Follows an non-exhaustive list of cases:

- If the syntax element is simple (fixed sized element, without any condition on its existence), only one action is created
- If the syntax element has some condition on its existence, three actions will be created: the first action tests if this element exists, the second action consumes the tokens relative to this syntax element and the third action is created for jumping to the next syntax in case of this element does not exists.
- If the syntax element must be repeated several times, three actions are created. An action is needed to check if this element needs to be repeated, an action which consumes the token of the syntax element and an action which is used to jump to the next syntax in case of the element must be not repeated anymore.
- If the parser actor needs to communicate with an external actor to parse this syntax element, then several actions are created for establishing a communication protocol between these two actors (see section 4.1).

The third step consists of building the Finite State Machine (FSM) of the final CALML parser. A preliminary sub-step is performed in order to build an intermediate tree which is

hal-00407949, version 1 - 28 Jul 2009

a more convenient representation of the initial tree so that it is then easier to perform the transformation for building the FSM. It consists of having a flatten representation of the relations between all the actions in order to have a better view on how the actions follows from each others.

Finally, the last step is to set the priorities between actions in case where they are more than one fireable actions at a given state of the actor.

Figure 6 shows a very simple example of BSDL description. The bitstream is composed of four elements: the first element is a 1-bit long element and is output on port A of the parser. The second element is 2-bits long and output on port B, third element is 3-bits long and output on part C and the fourth element is 4-bits long and output on port D. Figure 7 shows the example of CALML code generated for decoding the first syntax element. Figure 8 illustrates how the scheduling of the actions inside the parser.

```
    <xsd:element name="bitstream_root">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="firstelement" type="bs1:b1" rvc:port="A"/>
        <xsd:element name="secondelement" type="bs1:b2" rvc:port="B"/>
        <xsd:element name="thirdelement" type="bs1:b3" rvc:port="C"/>
        <xsd:element name="fourthelement" type="bs1:b4" rvc:port="D" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
```

**Fig. 6** Simple example of BSDL description.

```
<Action>
 <QID name="firstelement.read">
  <ID name="firstelement"/>
  <ID name="read"/>
 </QID>
 <Input kind="Elements" port="bitstream">
  <Decl kind="Input" name="b"/>
  <Repeat>
   <Expr kind="Var" name="BS1_B1_LENGTH"/>
  </Repeat>
 </Input>
 <Output port="A">
  <Expr kind="Var" name="b"/>
   <Repeat>
    <Expr kind="Var" name="BS1_B1_LENGTH"/>
   </Repeat>
 </Output>
</Action>
```

**Fig. 7** Example of source code: action for decoding ”firstelement” syntax element.

## 4.1 The case of unsized elements of syntax

As seen in the previous section, the parser is not only a simple actor that “demultiplexe” the raw data contained in the bitstream but runs also some parsing algorithms to decode some part of it. Unfortunately, automatizing the generation of these algorithms in CAL is

hal-00407949, version 1 - 28 Jul 2009

```
<Schedule kind="fsm" initial-state="root.firstelement_exists">

<Transition from="root.firstelement_exists" to="root.secondelement_exists">
 <ActionTags>
  <QID name="firstelement.read">
   <ID name="firstelement"/>
   <ID name="read"/>
  </QID>
 </ActionTags>
</Transition >

<Transition from="root.secondelement_exists" to="root.thirdelement_exists">
 <ActionTags>
  <QID name="secondelement.read">
   <ID name="secondelement"/>
   <ID name="read"/>
  </QID>
 </ActionTags>
</Transition >

[...]
</Schedule>
```

**Fig. 8** Example of source code: Finite State Machine of the parser

challenging because the bitstream schema is only a list of elements of syntax contained in the bitstream and does not indicate how to decode them. With such purpose in mind a mechanism has been set up in order to establish a communication between the parser and external FUs which will decode these parts of the bitstream. It is the case when decoding Variable Length Codes (VLC). Figure 9 illustrates the idea.

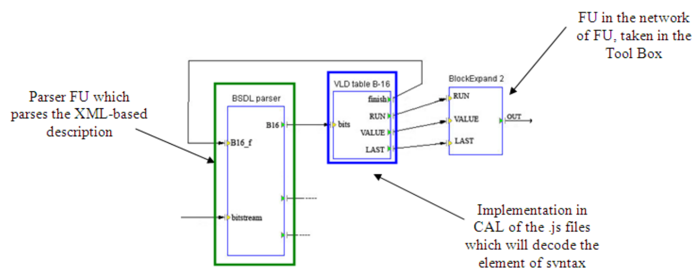


**Fig. 9** The data flow connections between the VLD table implemented as a Functional Unit and the parser.

During the validation phase, these algorithms were implemented in Javascript. In the code generation phase, these algorithms are implemented in FUs written in CAL. Currently, these FUs are not generated automatically and must be written manually.. However, in case of the Variable Length Decoding, a systematic procedure has been established in order to generate a part of these FUs in CAL from the VLD tables. The reader can refer to [14] for further details.

A communication protocol has been set up in the parser in order to communicate with the external FU capable of decoding the unsized elements of syntax. Each time a syntax element is met by the parser, the parser fires a `xxxx.read` action. When the parser meets a unsized syntax element (e.g. variable length codes), the parser fires a set of actions which are

hal-00407949, version 1 - 28 Jul 2009

necessary to communicate with the external Functional Units: `xxxx.read` to read the bit from the input port and to send the bit the the FU implementing the parsing algorithm, and `xxxx.finished` / `xxxx.notfinished` to decide if the parsing algorithm is finished or not and if the parser must send an additional bit. An example of code is shown on figure 10. The example illustrates the case of Variable Length Decoding.

## 5 Conclusion

Syntax parsing is a complex part in the video coding technology. Its complexity derive from the fact that the bitstream is composed of a large number of elements of syntax. The presence of some elements is conditioned by the value of other elements of syntax previously decoded. Writing a parser each time the syntax of the bitstream changes is a burdensome tasks for designers. It is often the case when developing new standards and new coding tools within MPEG. Thus, this paper presented a solution to the problem based on a systematic methodology for the validation of bitstream schemas (in BSDL) describing the syntax of bitstreams. The paper emphases a proof of concept of the methodology – starting from the validation of the BSDL schema and the automatic generation of CALML parsers – based on fully tested and proven standards (BSDL, XSL Transformation…). After these two stages (validation and generation), the CALML parser needs to be transformed in CAL form in order to be integrated with the Abstract Decoder Model within the MPEG Reconfigurable Video Coding (RVC) framework.

## References


1. ISO/IEC FDIS 23001-4, Maui, HI , US, *MPEG systems technologies – Part 4: Codec Configuration Representation*, 2009.
2. J. Eker and J. Janneck, "CAL Language Report," ERL Technical Memo UCB/ERL M03/48, 2003.
3. International Standard ISO/IEC FDIS 23001-5, *MPEG systems technologies - Part 5: Bitstream Syntax Description Language (BSDL).*
4. ISO/IEC FDIS 23002-4, Maui, HI , US, *MPEG video technologies – Part 4: Video tool library*, 2009.
5. "Open DataFlow Sourceforge Project," http://opendf.sourceforge.net/.
6. *ISO/IEC14496 Coding of audio-visual objects*. 2004.
7. 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.
8. 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.
9. 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.
10. A. Eleftheriadis, "Flavor: A Language for Media Representation," *ACM Int'l Conf. on Multimedia*, pp. 1–9, 1997.
11. D. Hong and A. Eleftheriadis, "XFlavor: bridging bits and objects in media representation," 2002.
12. 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.
13. 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.
14. 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.


hal-00407949, version 1 - 28 Jul 2009

```
<Action>
   <QID name="dct_dc_size_L.read">
      <ID name="dct_dc_size_L"/>
      <ID name="read"/>
   </QID>
   <Input kind="Elements" port="bitstream">
      <Decl kind="Input" name="b"/>
   </Input>
   <Output port="size_L">
      <Expr kind="Var" name="b"/>
   </Output>
   <Stmt kind="Assign" name="bit_number">
      <Expr kind="BinOpSeq">
         <Expr kind="Var" name="bit_number"/>
         <Op name="+"/>
         <Expr kind="Literal" literal-kind="Integer" value="1"/>
      </Expr>
   </Stmt>
</Action>
<Action>
   <QID name="dct_dc_size_L.notFinished">
      <ID name="dct_dc_size_L"/>
      <ID name="notFinished"/>
   </QID>
   <Input kind="Elements" port="size_L_f">
      <Decl kind="Input" name="f"/>
   </Input>
   <Guards>
      <Expr kind="BinOpSeq">
         <Expr kind="Var" name="f"/>
         <Op name="="/>
         <Expr kind="Literal" literal-kind="Integer" value="0"/>
      </Expr>
   </Guards>
</Action>
<Action>
   <QID name="dct_dc_size_L.finish">
      <ID name="dct_dc_size_L"/>
      <ID name="finish"/>
   </QID>
   <Input kind="Elements" port="size_L_f">
      <Decl kind="Input" name="f"/>
   </Input>
   <Input kind="Elements" port="size_L_data">
      <Decl kind="Input" name="data"/>
   </Input>
   <Guards>
      <Expr kind="BinOpSeq">
         <Expr kind="Var" name="f"/>
         <Op name="="/>
         <Expr kind="Literal" literal-kind="Integer" value="1"/>
      </Expr>
   </Guards>
   <Stmt kind="Assign" name="dct_dc_size_L">
      <Expr kind="Var" name="data"/>
   </Stmt>
</Action>
[...]
<Transition from="dct_dc_size_L1_exists" to="dct_dc_size_L1_result">
      <ActionTags>
         <QID name="dct_dc_size_L.read">
            <ID name="dct_dc_size_L"/>
            <ID name="read"/>
         </QID>
      </ActionTags>
</Transition>
<Transition from="dct_dc_size_L1_result" to="dct_dc_size_L1_exists">
      <ActionTags>
         <QID name="dct_dc_size_L.notFinished">
            <ID name="dct_dc_size_L"/>
            <ID name="notFinished"/>
         </QID>
      </ActionTags>
</Transition>
<Transition from="dct_dc_size_L1_result" to="next_syntax_element">
      <ActionTags>
         <QID name="dct_dc_size_L.finish">
            <ID name="dct_dc_size_L"/>
            <ID name="finish"/>
         </QID>
      </ActionTags>
</Transition>
```

**Fig. 10** Example of generated code in case of unsized syntax element

hal-00407949, version 1 - 28 Jul 2009