J Sign Process Syst (2011) 63:215–225
DOI 10.1007/s11265-009-0395-7

Automatic Synthesis of Parsers and Validation of Bitstreams Within the MPEG Reconfigurable Video Coding Framework

Christophe Lucarz · Jonathan Piat · Marco Mattavelli

Received: 23 January 2009 / Revised: 27 April 2009 / Accepted: 15 June 2009 / Published online: 9 July 2009

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 to the possibility of further improvements of such monolithic standards. The MPEG Reconfigurable Video Coding (RVC) framework is a new ISO/IEC 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.

C. Lucarz (✉) · M. Mattavelli
Microelectronic Systems Laboratory,
École Polytechnique Fédérale de Lausanne,
Lausanne, Switzerland
e-mail: christophe.lucarz@epfl.ch

M. Mattavelli
e-mail: marco.mattavelli@epfl.ch

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

Keywords Reconfigurable video coding · RVC · MPEG · Syntax parsing · Dataflow models · CAL

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, Advanced Video Coding (AVC) and Scalable Video Coding (SVC) have been developed and 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, whereas 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.

1.1 Essential Concepts in RVC

The essential concepts of the RVC framework can be summarized as follows:

- 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. A 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

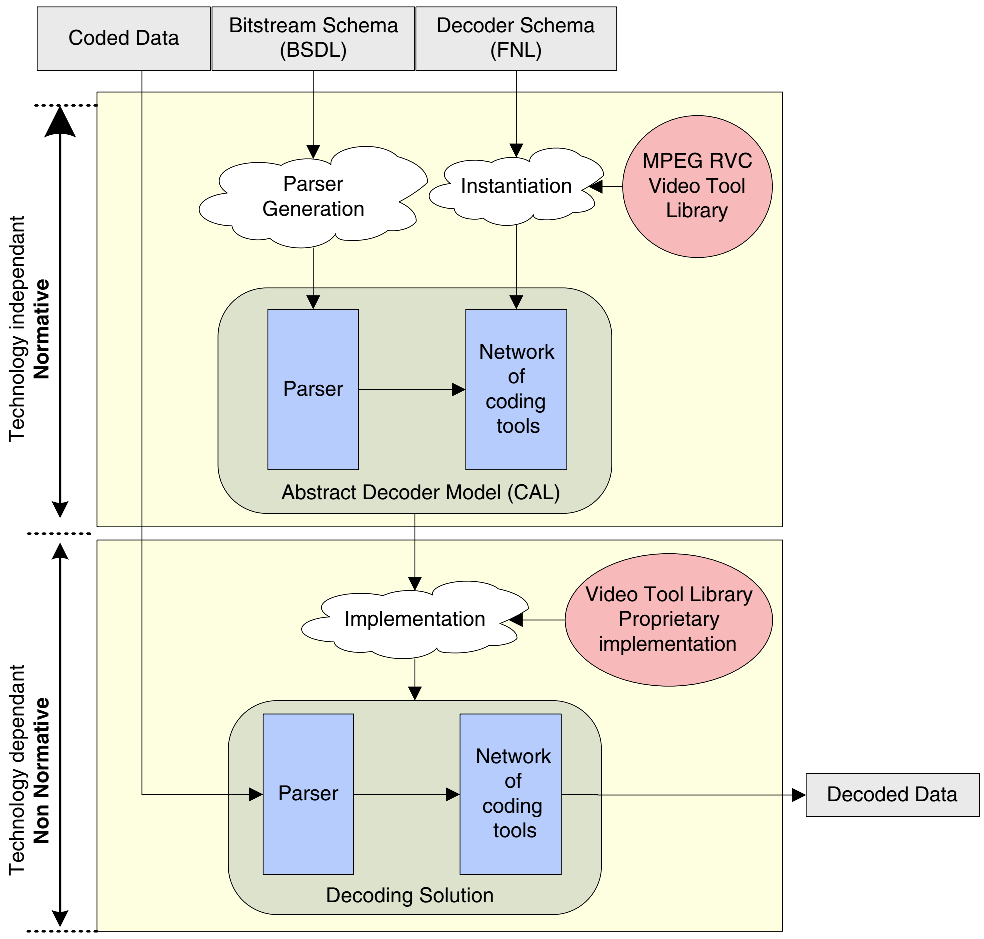

Figure 1 The reconfigurable video coding framework.

(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 Problem Definition

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 in general results to be a burdensome tasks. Moreover, it is certainly not an appropriate an efficient approach to write parsers 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 Stylesheet Language (XSL) Transformation is described in the second part of the paper. In any case, the validation of a bitstream description (written by hand or automatically generated) is the necessary preliminary step. Such procedure 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 for Defining Bitstream Syntaxes

ISO/IEC MPEG-B Part 5 is the standard that specifies BSDL [3] (Bitstream Syntax Description Language), a XML dialect describing generic bitstream syntaxes. For

Figure 2 A BSD and the corresponding bitstream schema is BSDL (**a**, **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 instance the description using BSDL of MPEG-4 AVC [6] bitstreams, represents in a compact form all the possible bitstream structures that conforms to MPEG-4 AVC syntax. A Binary Syntax Description (BSD) is a unique instance among all possible instantiations of a BSDL description. Such description represents a single MPEG-4 AVC encoded bitstream. It is no longer constituted by a BSDL schema, but by a XML file containing the data of the bitstream. Figure 2a shows a BSD associated to the corresponding BSDL schema shown in Fig. 2b.

An video bitstream is composed by a sequence of binary elements of the syntax having different lengths. Some elements are composed by a single bit, whereas others may contain several bits. The Bitstream Schema (in BSDL) indicates the length of such binary elements in a human and machine-readable format (hexadecimal, integers, strings ...). For example, hexadecimal values are used for start codes as shown in Fig. 2a. 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 specify and describe video bitstream syntaxes. Thus, BSDL schemas must specify all the elements of syntax, from the top hierarchy level down to the lowest level of syntax elements. Before the adoption of BSDL in the RVC framework, the existing BSDL descriptions were used to described scalable contents only at the high level of hyerarchy. Figure 2a is an example of BSDL description for MPEG-4 AVC video.

The choice of the language describing the syntax of bitstream is discussed in [8]. As result, BSDL has been preferred over Flavor and XFlavor [10, 11] for the following reasons:

- it is a stable language 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 is not used by the RVC framework. Thus, the XML-based BSDL description of the bitstreams has been preferred because it does not introduce a change of paradigm within the same framework.
- the XML-based description of BSDL better interacts with the XML-based representation of the configuration of RVC decoders (FU Network Descriptions) and the XML-based infrastructure of the existing tools.

3 BSDL Bitstream Schema Validation

Before generating parsers from bitstream schemas, it must be guaranteed that the schemas are correct, i.e. the schema correctly reflects the structure of the compressed data which is sent in it. If there is no validation procedure, that guarantees that the bitstream is exactly structured 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 correct 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 correspond to the schema under evaluation. If such procedure is repeated on a set of input bitstreams which are composed by all components of BSDL schema, the schema is considered as correct.

3.1 The Case of Unsized Elements of Syntax

In several cases, the size in bits of a syntax element is not known neither at compile-time nor at run-time

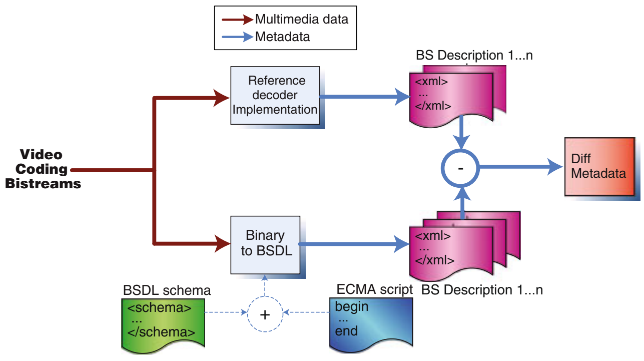

Figure 3 Illustration of the Bitstream Schema validation procedure.

(e.g. Variable Length Codes). The size in bits of the syntax element is known only during the decoding process of such syntax element. The validation process implies the generation of a BSD from a given bitstream. It means that the size of every syntax element must be determined. For the elements whose size is known *a priori*, generating the BSD is straightforward. Conversely, for the elements whose size is not known *a priori*, a parsing algorithm has to be implemented in order to determine the size of such elements of syntax. For the validation process, such 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. The procedure enables the specification of parsing algorithms in a bitstream schema that can be used by BintoBSD, thus enabling the processing of data structures that cannot be determined only by BSDL constructs.

Figure 4a 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 Fig. 4b.

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 bitstream information. The implementation of BintoBSD provides the ECMAScript with functions enabling to:

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

Therefore 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 local name as fragment identifier to the namespace.

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

Figure 4 Call and example of a parsing algorithm in Javascript (**a**, **b**).

```
<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

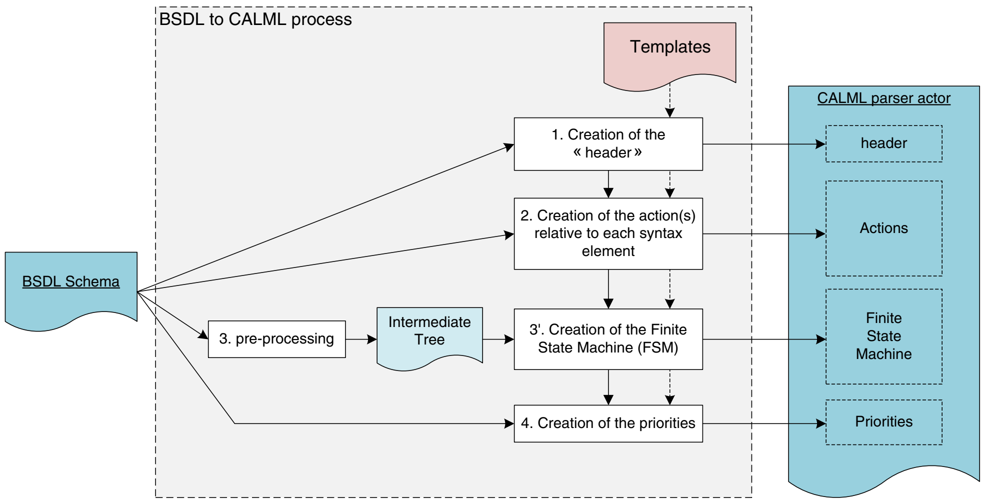
BSDL to CALML process
Templates
1. Creation of the « header »
2. Creation of the action(s) relative to each syntax element
BSDL Schema
3. pre-processing
Intermediate Tree
3'. Creation of the Finite State Machine (FSM)
4. Creation of the priorities
CALML parser actor
header
Actions
Finite State Machine
Priorities

Figure 5 Block diagram of the XSL Transformation process.

4 Synthesis of Parsers from Bitstream Descriptions

The bitstream structure is described in BSDL, a XML dialect (Section 2). The bitstream first needs to 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 appropriate procedures 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. The challenge is to develop transformations such that the resulting executable (the parser) is capable of handling all the legal combinations of the BSDL constructs constituting the schema.

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

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

The BSDL to CALML transformation is composed by four main steps. At each step of the process, the BSDL schema is analyzed 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 presents some conditions 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 syn-

Figure 6 Simple example of BSDL description.

```
  <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>
```

Figure 7 Example of source code: action for decoding "firstelement" syntax element.

```
<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>
```

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

```
<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>
```

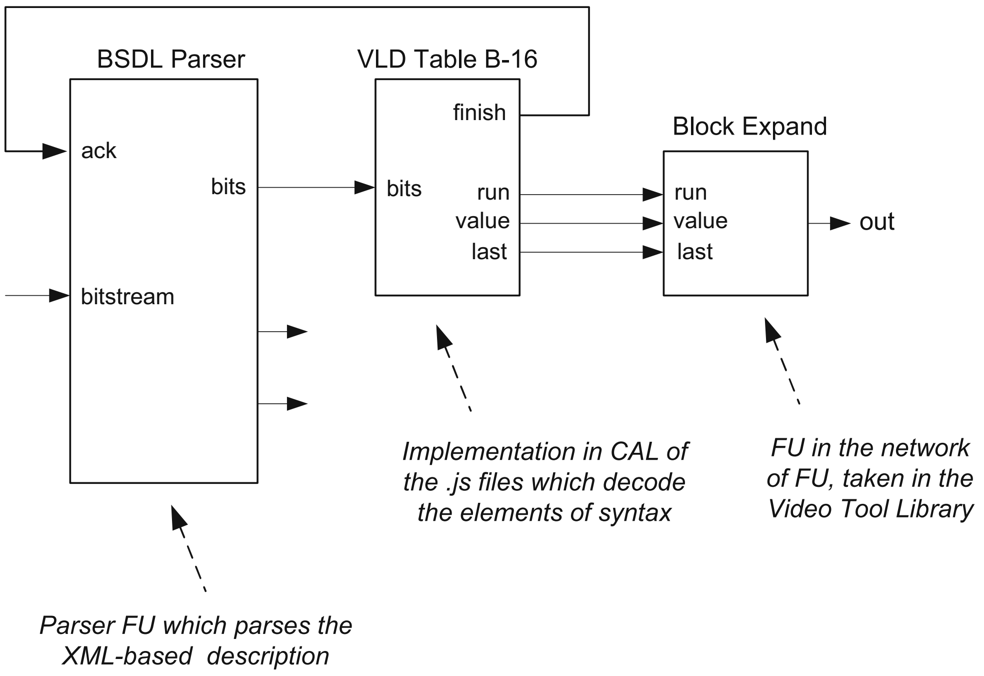
BSDL Parser
ack
bits
bitstream
VLD Table B-16
finish
bits
run
value
last
Block Expand
run
value
last
out
Implementation in CAL of the .js files which decode the elements of syntax
FU in the network of FU, taken in the Video Tool Library
Parser FU which parses the XML-based description

Figure 9 The data flow network composed by the VLD table decoding processes implemented as a Functional Unit and the bitstream parser.

tax element and 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.

Figure 10 Example of generated code in case of unsized syntax element.

```
<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>
```

– 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 a more convenient representation of the initial tree so that it is then easier to perform the transformation for building the FSM. The process consists of obtaining 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 there exist more than one fireable action at a given state of the actor.

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

4.1 The Case of Unsized Elements of Syntax

As seen in the previous section, the parser is not only a simple actor that "demultiplexes" the raw data contained in the bitstream, but also executes parsing algorithms to decode some sections of the bitstream. Unfortunately, automatizing the generation of these algorithms in CAL is complex because the bitstream schema is only a list of elements of syntax contained in the bitstream and does not indicate how to decode them. Such procedure is related to the semantic of the element and need to be known by the parser. The mechanism set up in the RVC framework consist of establishing a communication between the parser and external FUs which provide the implementation capable of decoding these sections of the bitstream. This is the case when decoding Variable Length Codes (VLC) elements for which VLD tables are the associated decoding procedures are embedded in library components that correspond to the semantic of the unsized element. Figure 9 illustrates the CAL network composed by the synthesized parser and the external CAL actors (i.e. FUs) performing the decoding of unsized syntax elements.

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 have to be written manually. However, in case of the Variable Length Decoding, a systematic procedure has been developed in order to generate these FUs in CAL directly from the VLD tables. The reader can refer to [14] for further details of this process.

A communication protocol need to be defined 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 determined by the parser, the parser fires a `xxxx.read` action. When the parser finds an unsized syntax element (e.g. variable length codes), the parser fires a set of actions which are 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 terminated or not and if the parser must send an additional bit. An example of code is shown on Fig. 10. The example illustrates the case of Variable Length Decoding.

5 Conclusion

Syntax parsing is a complex component of video coding technology. The complexity derive from the fact that the bitstream is composed of a large number of hierarchical elements of syntax. The presence of some elements is conditioned by the value of other elements of syntax previously decoded and processed. Writing a new 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 provides 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 so as to be integrated within the

Abstract Decoder Model that specify a decoder configuration in the MPEG Reconfigurable Video Coding (RVC) framework.

References

1. ISO/IEC FDIS 23001-4 (2009). *MPEG systems technologies—part 4: Codec configuration representation*. Maui.
2. Eker, J., & Janneck, J. (2003). *CAL language report.* ERL Technical Memo UCB/ERL M03/48.
3. International Standard ISO/IEC FDIS 23001-5 (2005). *MPEG systems technologies—part 5: Bitstream syntax description language (BSDL)*.
4. ISO/IEC FDIS 23002-4 (2009). *MPEG video technologies—part 4: Video tool library*. Maui.
5. Sourceforge (2009). Open dataflow sourceforge project. http://opendf.sourceforge.net/.
6. ISO/IEC14496 (2004). Coding of audio-visual objects.
7. Lucarz, C., Mattavelli, M., Thomas-Kerr, J., & Janneck, J. (2007). Reconfigurable media coding: A new specification model for multimedia coders. In *IEEE workshop on signal processing systems* (pp. 481–486).
8. Thomas-Kerr, J., Janneck, J., Mattavelli, M., Burnett, I., & Ritz, C. (2007). Reconfigurable media coding: Self-describing multimedia bistreams. In *IEEE workshop on signal processing systems SiPS 2007*. Shanghai, China, 17–19 April 2007.
9. Thomas-Kerr, J., Burnett, I., Ritz, C., Devillers, S., De Schijver, D., & Van de Walle, R. (2007). Is that a fish in your ear? A universal metalanguage for multimedia. *IEEE Multimedia, 14*(2), 72–77.
10. Eleftheriadis, A. (1997). Flavor: A language for media representation. In *ACM int'l conf. on multimedia* (pp. 1–9).
11. Hong, D., & Eleftheriadis, A. (2002). *XFlavor: Bridging bits and objects in media representation.*
12. Janneck, J. W., Miller, I. D., Parlour, D. B., Mattavelli, M., Lucarz, C., Wipliez, M., et al. (2008). Translating dataflow programs to efficient hardware: An MPEG-4 simple profile decoder case study. In *Design, automation and test in Europe (DATE)*. Munich, Germany.
13. Wipliez, M., Roquier, G., Raulet, M., Nezan, J.-F., & Déforges, O. (2008). 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.
14. Li, J., Ding, D., Lucarz, C., Keller, S., & Mattavelli, M. (2008). Efficient data flow variable length decoding implementation for the MPEG reconfigurable video coding framework. In *IEEE workshop on signal processing systems*. Washington DC.

Christophe Lucarz received his M.Sc. degree in Electrical Engineering from the Institut National des Sciences Appliquées (INSA Lyon, France) in 2006. He is currently a researcher at the "Multimedia Architectures Research Group" at the Ecole Polytechnique Fédérale de Lausanne (EPFL - Switzerland) and is working towards his Ph.D. degree. His research topic is about high level design space exploration of complex digital systems. It includes application programming (dataflow models), the scheduling/partitioning problem (combinatorial optimization) and the implementation issue (code generation). He is also taking part in the MPEG ISO/IEC standardization committee in video coding and the ACTORS European Project.

Jonathan Piat is a PhD student at the National Institute of Applied Sciences of Rennes (INSA) and a member of the IETR laboratory in Rennes. He received his postgraduate certificate in Distributed Software Architecture from the MINE school of Nantes in 2007 and computer engineering from the Polytechnic School of the University of Nantes in 2007. His main research interests include dataflow-models, Computer Aided software Design, multiprocessor rapid prototyping, and video coding/decoding.

Marco Mattavelli was born in Milano, Italy, he received his Diploma of Electrical Engineering from the Politecnico di Milano. Since 1988 he joined the "Philips Research Laboratories" of Eindhoven in the framework of EUREKA-95 (HDMAC) project. Main research activities regarded channel and source coding for optical recording, electronic photography and signal processing of TV and HDTV signals. Since October 1991 he joined the "Signal Processing Laboratory" (LTS) of the "Swiss Federal Institute of Technology" (EPFL) where he got his PhD in 1996. Then, he has been involved in several research projects and didactic activities. He has been very active in the last 10 years in the ISO/IEC standardization activities (known as MPEG), for which he has been Chairman of the Implementation Study Group. For his work he received the ISO/IEC Award in 1997 and 2003. He is currently leading the "Multimedia Architectures Research Group" in the "Laboratory of Microelectronic Systems" of EPFL. His major research activities include: methodologies for high level specification and modeling of complex systems, architectures and systems for video coding, high speed image acquisition and video processing, applications of combinatorial optimization to signal processing. He holds patents in the multimedia and video processing fields. He is the author of more than 100 publications and has served as invited editor for several conferences and scientific journals.

Marco Mattavelli started his research activity at the "Philips Research Laboratories" of Eindhoven in 1988 on channel and source coding for optical recording, electronic photography and signal processing of HDTV. In 1991 he joined the "Swiss Federal Institute of Technology" (EPFL) where he got his PhD in 1996. He has been a chairman of a sub group of MPEG ISO/IEC standardization committee. For his work he received the ISO/IEC Award in 1997 and 2003. He is currently leading the "Multimedia Architectures Research Group" at EPFL. His current major research activities include methodologies for specification and modeling of complex systems, architectures for video coding, high speed image acquisition and video processing systems, applications of combinatorial optimization to signal processing. He is the author of more than 100 publications and has served as invited editor for several conferences and scientific journals.