RECONFIGURABLE MEDIA CODING: SELF-DESCRIBING MULTIMEDIA BITSTREAMS

Joseph Thomas-Kerr[1], *Jorn Janneck*[2], *Marco Mattavelli*[3], *Ian Burnett*[1] *and Christian Ritz*[1]

[1]University of Wollongong, [2]Xilinx, [3]Ecole Polytechnique Federale de Lausanne (EPFL)

ABSTRACT

The development of MP3 and JPEG sparked an explosion in digital content on the internet. These early encoding formats have since been joined by many others, including Quicktime, Ogg, MPEG-2 and MPEG-4, which poses an escalating challenge to vendors wishing to develop devices that interoperate with as much content as possible. This paper presents aspects of Reconfigurable Media Coding (RMC), a project currently underway at MPEG to define a self-describing bitstream format. In other words, an RMC bitstream contains metadata to assemble a decoder from a fundamental building-blocks, as well as a schema that describes the syntax of the content data, and how it may be parsed. RMC makes it easy to extend (reconfigure) existing codecs, for example adding error resilience or new chroma-subsampling profiles, or to build entirely new codecs. This paper addresses the *bitstream syntax* component of RMC, validating the approach by applying it to the recent MPEG-4 Video simple profile coder.

1. INTRODUCTION

The MP3 digital audio format was first published in 1991. Only in the last five years, however, has Moore's law allowed such audio to be decoded by battery-powered, portable devices, fundamentally changing the way most people obtain and consume music, and making MP3 a household name. In a similar period, MP3 has been joined by a plethora of other multimedia formats: Windows Media, Quicktime, Ogg, Flash, MPEG-2, and MPEG-4 to name a few. Furthermore, the diversity of the devices on which multimedia content is rendered, and of the communication channels across which it is delivered, has increased dramatically. This proliferation of multimedia formats and devices presents an escalating challenge to interoperability between the format that content is stored in and the devices on which users which to consume it.

Despite the recent growth in multimedia coding technologies, the process of standardizing new algorithms and coding techniques remains very lengthy. This standardization process has been necessary for coding technology to be useful to the wider public, because without it there is no guarantee that one vendor's encoder will work with another's decoder. The typical lead-time between innovation and mass-deployment for recent standards has been three to five years; work on H.264/AVC [1] for example began in 2002, but did not see significant use until the release of the video iPod in late 2005.

This paper presents aspects of Reconfigurable Media Coding (RMC), an alternative paradigm for coders that greatly simplifies interoperation between increasingly diverse multimedia devices. This paradigm makes content *self-describing*, in that an RMC bitstream includes information to build a decoder from fundamental building-blocks (Figure 1). As a result, multimedia decoder vendors no longer need to (largely independently) develop implementations of new coding formats for their devices. Instead, the device will provide a generic RMC decoder which can be reconfigured on-the-fly according to the information in an RMC bitstream. RMC is currently in the process of standardization by MPEG [2].

What follows is a discussion of the usage scenarios (section 1.1) and requirements (1.2) for RMC. The remainder of the paper will give particular emphasis to the *syntax description* component of the work, considering alternative approaches (section 2), the programming paradigm used to allow reconfigurability (section 3), and the syntax description language itself (section 4). See [3] for a general treatment of RMC.

1.1. Usage scenarios for reconfigurable coding

Media bitstreams can describe the decoders required to process them in several ways, which differ in the tradeoffs they make with respect to, for instance, generality, processing requirements, openness, and infrastructure. For instance, in a *library-based* decoder, the bitstream describes its decoder as a network that consists of the instantiation (and parametrization) of decoding tools taken from a library of predefined modules. This approach results in relatively small configuration overhead, but it assumes the existence of a standardized or otherwise agreed-upon library of decoder modules, or a mechanism by which a platform may acquire new modules on the fly (e.g. downloading them over a network).

At the other end of the spectrum are *fully programmable* decoders, in which the bitstream contains a complete exe-

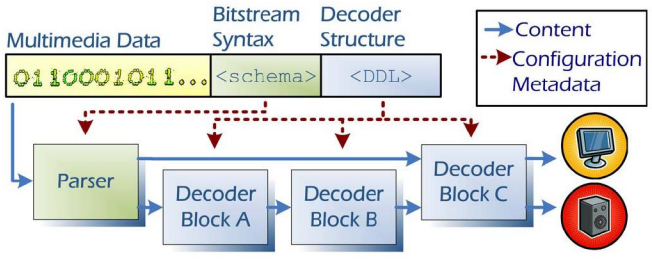

Fig. 1. A RMC bitstream is *self-describing*

```
VideoObjectLayerType = ( longHeader | shortHeader ),
                       { VideoObjectPlaneType }
longHeader = VOLStartCode, (*...and so on*)
VOLStartCode = StartCodeType
StartCodeType = 4 * hex-digit
hex-digit = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"
          |"A"|"B"|"C"|"D"|"E"|"F"
```

$(a) EBNF$

```
VideoObjectLayerType ::= SEQUENCE {
            header  VOLHeaderType,
            vops    SEQUENCE OF vop}
VOLHeaderType ::= CHOICE {
            longHeader LongHeaderType,
            shortHeader ShortHeaderType}
LongHeaderType ::= SEQUENCE {
            volStartCode StartCodeType,–...and so on–}
StartCodeType ::= OctetStringType (SIZE(4))
```

$(b) ASN.1$

```
class VideoObjectLayer() {
  int(32)* next_bits; //look-ahead only
  if(next_bits == 0x00000120) {
    short_video_header = 0;
    int(32) VOLStartCode;
    // ... and so on ...
  } else {
    short_video_header = 1;
    // ...
  }
  do {
    VideoObjectPlane();
    int(32)* next_bits;
  } while (next_bits ==  0x000001B6);
}
```

$(c) Flavor$

```
<complexType name="VideoObjectLayerType"rmc:port="vol">
 <sequence>
  <choice>                                                <!– a –>
   <group ref="longHeader" bs2:ifNext="00000120"/>
   <group ref="shortHeader"/>
  </choice>
  <element name="VOP" type="VideoObjectPlaneType"
           bs2:ifNext="000001B6" maxOccurs="unbounded"/>
 </sequence>
</complexType>
<group name="longHeader">
 <sequence>
  <element name="VOLStartCode" type="SCType"/>
   <!– ...and so on... –>
   <bs2:variable name="mbCount" value="(($volWidth+15)
                 idiv 16)*(($volHeight+15) idiv 16)"/><!– b –>
 </sequence>
</group>
```

$(d) BSDL$

Fig. 2. There are numerous syntax description metalanguages. These examples show a small part of the MPEG-4 Video syntax

cutable description of its decoder in a platform independent specification. This scenario requires much more infrastructure on the decoder side, which needs to be able to quickly translate a decoder specification into an efficiently executable implementation on its specific hardware platform. It would thus need to incorporate an complete compilation infrastructure for the decoder specification language. That language, in turn, needs to be platform-agnostic, and still yield reasonable implementations on a wide range of hardware and software targets. In such a scenario, bitstreams may describe decoders that are arbitrarily tuned to their specific requirements, without a for encoder and decoder to agree on a specific library.

Somewhere between the two are *hybrid* decoders, in which some of the coding tools could be specified using an executable language, while others are instantiated from a standard library. A plausible instance of this would be the executable description of the bitstream parser (for instance in the form of a grammar which is interpreted or compiled on the fly), but standard blocks for the remaining decoder modules.

1.2. Requirements for reconfigurable coding

The key requirement for reconfigurable decoders is that their basic architecture allows for a variety of implementations. This may be, for example, in software on single or multiple processors, in hardware, or in a heterogeneous mix of hardware and software components. Consequently, the description of such a decoder should lend itself easily to parallelization, and it should permit the use of various scheduling policies.

This points to the need for a component model that emphasizes strong encapsulation of state and thin communication interfaces. In particular, the requirement for parallelizability, and schedule independence suggests the absence of shared memory between components. In the absence of shared memory, components need to interact by sending each other messages containing packets of data we call *tokens*.

These requirements outlined are usually met very well by approaches known by names such as *dataflow* or *stream processing*, which include Kahn process networks [4] and Dennis Dataflow [5]. The RMC work builds on CAL [6] for describing modules of media codecs. It is a language for writing dataflow blocks, designed to combine expressiveness with analyzability. For further discussion of the various stream processing approaches and their applicability to RMC, see [3].

Finally, a reconfigurable decoder requires information about the syntax of the media content, so that it may pass the correct input data to each of the subsequent components. This information must include enough detail to parse data into the atomic units expected by each component. It must identify not just cardinality constraints on syntactical elements, but also the algorithm to determine the actual cardinality of an instance. Alternatives for this task are discussed below.

2. APPROACHES TO SYNTAX DESCRIPTION

Syntax description is a mature field that has its roots largely in programming language specification. The (Enhanced) Backus-

Naur Form (EBNF) [7] is a notation that has become the de facto standard for specifying the syntax of programming languages, although it has many variants. Syntax is, in fact, a fundamental aspect of almost any form of communication, and alternative syntax notations have been developed for other domains. For example, Abstract Syntax Notation One (ASN.1) [7] is widely used to specify network protocols, and XML Schema or Document Type Definitions constrain the syntax of XML documents [7].

Specification of multimedia syntax, on the other hand, has traditionally used ad hoc notations (Quicktime, for example [8]), some of which are loosely based on EBNF (such as AVI [9]). More recently, two multimedia-specific syntax metalanguages have been proposed: Flavor and BSDL. Flavor [10] uses C++/Java-like expression to specify bitstream syntax for automatic parser generation. The Bitstream Syntax Description Language (BSDL) [11], on the other hand, is an extension of XML Schema, specifying how atomic data-types from the latter map to binary symbols, and providing control-flow constructs to manage parsing. BSDL was originally designed to enable adaptation of scalable multimedia content in a format-independent manner, that is, using adaptation software that did not possess detailed knowledge of the content format it was adapting. This was achieved via an XML *view* of the content, hence the choice of XML Schema.

To highlight the differences between syntax description languages, Figure 2 shows descriptions of the same part of an MPEG-4 Visual bitstream [12]. ASN.1 explicitly separates content (abstract syntax, shown in the figure) from encoding (not shown). This is in recognition of the fact that identical message content may be encoded differently depending on context (for example, a more efficient binary encoding may be preferred for low bandwidth applications). This separation is indeed valid for multimedia: the same content could be encoded in H.264, or in MPEG-4 Simple Profile, for example. However, the distinction is not relevant at the level of *individual symbols* at which it is made in ASN.1. For multimedia, abstraction of content from encoding at the symbol level significantly adds to complexity, without improving portability.

In order to parse raw content of a particular format, additional information is necessary beyond a description of its syntax. This may be seen, for example, in EBNF (Figure 2a) and ASN.1 2b), which specify that a VOL object may contain either a long header or a short header, but not how to tell which is actually present within a bitstream. Flavor provides this information via an `if` block, and BSDL using an `bs2:ifNext` attribute (or others, see Section 4.1). In the RMC framework, BSDL is preferred over Flavor because

- it is stable and defined by an international standard [11];
- its XML-based syntax integrates well with the XML syntax used to describe the rest of the RMC decoder; and
- the RMC bitstream parser may be easily derived by transforming the BSDL using standard tools (e.g. XSLT [13]).

3. DATAFLOW IN RECONFIGURABLE CODING

The stream-oriented programming paradigm of dataflow lends itself naturally to describing the processing of media streams that pervade most of media coding. In addition, the strong encapsulation afforded by the actor model provides a solid foundation for the modular specification of media codecs. *Actors* are the fundamental modules that are the basic building blocks of a dataflow system. As in Dennis dataflow [5], the actors we use perform their computation in a sequence of atomic steps (*firings*). In each of these steps, they can do any combination of the following:

- Consume one or more tokens at any of their input ports;
- Produce token(s) on output port(s); and/or
- Modify internal state variables;

The state variables of two different actors are always disjoint; one actor can never directly modify the state of another. It is this property that allows us to view actors exclusively in terms of the production and consumption of tokens on their ports, which in turn greatly facilitates their composition. As a side effect, the strict separation of state and the asynchronous communication through token sending and receiving allows for the distributed and parallel implementation of dataflow networks, which becomes especially useful for demanding media coding applications, such as those involving high-definition video.

The actors are described in an actor language called CAL, which is described here only in so far as is necessary for the subsequent discussion of parser generation. Further detail may be found in [6]. In essence, the description of an actor has to define the atomic steps it can make at each point in its execution. In CAL, these steps are described as *actions*. An actor description can comprise any number of actions. Each action consists of the elements of an atomic step (how much input is consumed, the output values produced, and how the state is modified), along with a definition of its enabling conditions (*guards*), which all need to be true for the action to be able to fire. The guards of an action always include the requirement that all tokens that will be consumed by the action need to be present. In addition, they may include the following:

- Any boolean predicate on the values of the input tokens, state variables, or a combination of the two; and/or
- The priority of the action with respect to other actions.

Priorities can be used to constrain the choice of the next action to be fired in cases where more than one action is enabled. If this happens, and there is no priority ordering between the concurrently enabled actions, then the next action may be either of them and the choice is unspecified. This kind of non-determinism may be desirable in some cases, but priorities can be used in other situations requiring deterministic execution.

One language element that is of particular relevance when building parsers is a finite state machine (FSM) which can be part of the actor state. Here, an action can be defined as the transition between any two states of the FSM. In this case,

```
<complexType name="VideoObjectPlaneType">
 <sequence>
  <element name="vopHdr" type="VOPHeaderType"
                cmc:port="vopHeader"/>                    <!– c –>
  <element name="motionShapeTexture" minOccurs="0"
    type="MotionShapeTextureType" bs2:if="$vopCoded=1"/>
 </sequence>                                              <!– d –>
</complexType>

<complexType name="VOPHeaderType">
 <sequence>
  <element name="vopSC" type="SCType"/>
  <!– ... –>
  <element name="vopCoded" type="bs1:b1"
                cmc:variable="true"/>                     <!– e –>
 </sequence>
</complexType>
<complexType name="MotionShapeTextureType">
 <sequence>
  <!– ... –>
  <element name="MB" type="MBType" cmc:port="MB"
       maxOccurs="unbounded" bs2:nOccurs="$mbCount" />
 </sequence>                                              <!– f –>
</complexType>

<complexType name="MBType">
 <sequence>
  <!– ... –>
  <element name="horizMVData" type="MVDataType"/>
  <!– ... –>
 </sequence>
</complexType>

<simpleType name="MVDataType">
 <restriction base="bs1:extensionType">                   <!– g –>
  <annotation><appinfo>
   <bs1:script ref="mvData.cal" lang="cal"/>
  </appinfo></annotation>
 </restriction>
</simpleType>
```

Fig. 3. More of the BSDL Schema for MPEG-4 Video

actions acquire an additional guard: that the FSM is in a state having a transition effected by the action in question.

4. BITSTREAM DESCRIPTION IN RMC

The syntax of the media content in an RMC bitstream is described by a BSDL schema that is delivered alongside the content, as shown in Figure 1. An RMC decoder transforms this schema into a parser block that converts the raw data into structured fields and objects which are used by subsequent decoder blocks. In the reference implementation described here, this is described using the XML Transformation language XSLT. However, implementations may use other means to parse an RMC bitstream according to its schema.

4.1. The Bitstream Syntax Description Language

Before looking in detail at the parser generation process, we will first expand upon the example of Figure 2(d) to highlight some pertinent features. BSDL is an extension of XML Schema [11]. It is the latter that defines the structural features of a BSDL Schema (known as *particles*): choice, sequence, all, element and group. Elements are given a type, which is defined by either a complexType (the element contains another particle), or a simpleType (the element contains binary content). BSDL defines how simple types are read from the bitstream. For example, the facet xsd:maxExclusive defines the number of bits read by integer types, and xsd:length the number of bytes in a string or hexBinary type. BSDL also adds annotations that provide the additional information required for bitstream parsing, including identification of which choice option is actually chosen, whether optional particles are in fact present, and how many occurrences of multiple particles exist. We have already seen one example of these annotations: the bs2:ifNext attribute in Figure 2(d), at marker (a). This attribute specifies that the structure to which it is attached should be read from the bitstream only if *the next bytes in the bitstream correspond to the hex value of the attribute*[1].

The other BSDL construct in Figure 2(d) is bs2:variable (at b), which, (unsurprisingly) creates a variable with the given name containing the result of the value expression. Variables are used in other annotation expressions within the schema. In this case, the variable mbCount stores the number of MacroBlocks [2] in each frame. The number of macroblocks in a frame are thus computed from the height and width (in pixels) of the video.

Figure 3 shows more of the BSDL Schema for MPEG-4 Video, highlighting other pertinent features of the language:

(c) rmc:port on the vopHdr element specifies that this structure (and descendant content) should be output from the parser on the vopHeader port.

(d) bs2:if is similar to bs2:ifNext except that it evaluates a boolean expression rather than a value in the raw bitstream.

(e) The variable $vopCoded is stored during the parsing of the VOPHeaderType, where rmc:variable is a shorthand variable declaration indicating that the name and value are equivalent to those of the parent element.

(f) bs2:nOccurs specifies the number of occurences of the macroblock element, using the variable computed in Figure 2(d). Macroblocks are also output to a port.

(g) Numerous fields in MPEG-4 or any other media format are encoded using variable-length codes to increase bandwidth efficiency. Common techniques such as Huffman coding or lookup tables could be hard-coded into the language, but this would necessitate normative changes to support future encoding methods, defeating the purpose of RMC. Consequently, arbitrary decoding algorithms may be specified in BSDL by sub-classing bs1:extensionType, providing a script node that implements the decoder. RMC uses the CAL language to specify the decoder operation, so this is the language used in BSDL scripts in RMC.

[1] bs2:ifNext has other options too; the interested reader is directed to [11].
[2] blocks of 16x16 pixels, the atomic unit in MPEG video codecs.

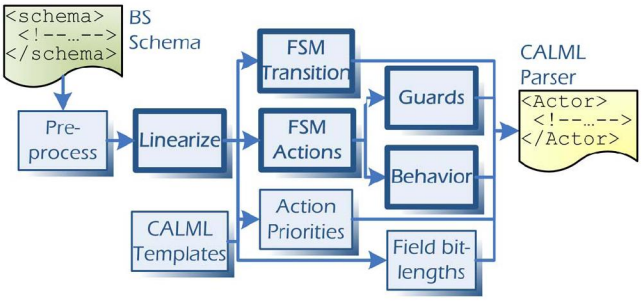

Fig. 4. Components of the parser generation process

4.2. Syntax Schemata in RMC

The BSDL Schema transmitted with an RMC bitstream contains all of the information necessary to parse the rest of the bitstream. The decoder translates this schema into a parser block whose task is to convert the raw bits into structured data that may be processed by subsequent decoder modules. Although this translation is relatively involved (Figure 4), the declarative model shared by both BSDL and CAL means that the translation process may be efficiently specified. Figure 4 shows the components of this process. Each component is implemented in a separate XSLT stylesheet, which is then imported by a master sheet that coordinates the overall process.

Preprocessing is the first operation conducted by the stylesheet. In general, a BSDL Schema may be composed of a number of separate Schemata, which are imported by a master document (much the same as the stylesheet). The preprocessing stage is therefore necessary to collect the individual Schemata into a single intermediate tree, taking care to correctly manage the namespace of each component Schema. The preprocessor also performs a number of other tasks, including assigning names to anonymous types and structures (so that they may be referred to by the FSM transition set), resolving inheritance relationships, and removing structures which are not significant to the parsing process.

Finite State Machine (FSM) design is the major component of the parser actor. The FSM schedules the reading of bits from the input bitstream into the fields in the various output structures, along with all other components of the actor. The FSM is specified as a set of transitions, where each transition has an initial state, a final state, and an action. Computing the FSM from a BSDL Schema has several components, each of which are highlighted in bold within the figure.

Actions scheduled by the FSM control the next-state decision mechanism via their **Guard** expressions, which are built from the control-flow constructs in the BSDL Schema (if, ifNext, nOccurs and length). The **Behaviour** of each action is to complete such tasks as storing data in the appropriate location in the output structure and/or variables, and setting the number of bits to be read for the subsequent field.

The state pattern of the FSM is predominantly linear: the first field is read, then the next field, then the next, and so on. Consequently, the hierarchical structure of the BSDL Schema must be converted into a linear sequence of read instructions, from which the FSM Transition and Action sets may be built. This operation is performed by the **Linearize** component, which reads the preprocessed BSDL Schema and outputs an intermediate data structure comprising the linear sequence of read actions. It is from this intermediate structure that the FSM is assembled.

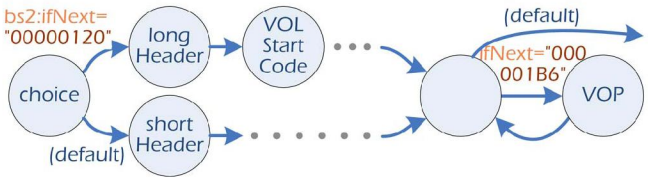

Fig. 5. FSM fragment for a choice particle

There are two exceptions to this linearity: Choice particles and Union types. Choice particles (such as that in Figure 2(d)) cause the FSM to diverge from its linear path to one of a number of parallel paths, each of which parse a single option of the choice. For example, Figure 5 depicts the FSM fragment for the VOL header choice of Figure 2(d). Each of the parallel paths has a test action that determines which of the options is selected. As before, the guards on each test action are built from control-flow constructs in the BSDL Schema. BSDL specifies that the order of options within a choice establishes their priority: the first option has priority over the second, and so on. These **priorities** are recorded in the actor as priorities between the test actions.

Union types are very similar to Choice particles, except that instead of choosing between a number of different objects to instantiate, a Union chooses between a number of different types that a single object could take. For example, a single field could be either 16 bits or 32 bits, depending on the resolution it is required to record. Union types have the same state structure as choice particles, but differ in the composition of their test guards.

Field bit-length in BSDL is specified indirectly via the xsd:maxExclusive facet of XML Schema. A stylesheet component is therefore required to compute the bit-length of simple types within the schema from their maxExclusive value. Once computed, the value is stored in a constant identified by the type name, and subsequently used whenever a field of that type is read from the bitstream.

Finally, the **CAL** component declares templates for each of the constructs in the language, such as an FSM schedule, a function call, or an assignment. These templates are called by other components of the stylesheet when building the actor. Collecting all of the CAL syntax into a single stylesheet also means that an alternative stylesheet could be provided in place of the CAL sheet, for example containing templates that output CALML (CAL-XML), or even unrelated languages.

5. RESULTS

This section presents an assessment of the suitability of BSDL for syntax description within the RMC framework. There are fundamentally two questions that must be addressed here:

- Is BSDL capable of describing real-world multimedia formats in complete detail?
- If so, can the resulting BSDL Schema be used efficiently by a generic parser module to read bitstreams? That is, does BSDL allow efficient RMC implementations?

There are also other secondary considerations such as readability (i.e. complexity), ease of development and debugging, and verbosity.

The MPEG-4 Video Simple Profile [12] standard is used as a test subject to address the questions raised above. Although RMC is designed for the creation of new codecs, it is prudent to first establish that it may be used successfully to describe existing ones. The MPEG-4 standard specifies syntax predominantly via tables of pseudocode, an example of which is shown in Table 1, although some parts are described in prose (such as DCT coefficient decoding), and others using look-up tables (typically VLCs). The pseudocode shown in the figure corresponds to the BSDL Schema extract shown in Figure 3. BSDL uses declarative structures (sequence, choice) rather than imperative constructs (for, do..while), but it is straightforward to develop a BSDL Schema based on the pseudocode.

BSDL is also able to express the DCT decoding process specified in prose in the MPEG-4 standard. In doing so, it in fact provides a significantly more objective specification mechanism than the standard itself. BSDL does not directly support VLC decoding. However, it provides an extension mechanism for such cases to allow the parsing process to be specified in an external language (in this case CAL). Using this, it is simple to express the MPEG-4 VLC tables in the BSDL Schema in a form that an RMC decoder is able to process.

A complete BSDL Schema for MPEG-4 Video Simple Profile has been developed and tested, verifying that BSDL is sufficiently descriptive to be used with real-world media. The use of BSDL in an RMC parser module is described in the previous section. Furthermore, work is currently in progress to extend this validation to H.264/AVC, and to demonstrate the extensibility of the RMC framework by deploying content encoded using 4:2:2 and 4:4:4 chroma subsampling: patterns that don't exist within the standardized version of MPEG-4. In an RMC bitstream, this means adding extra chroma blocks in each Macroblock, and changing the chroma block pattern header field. In the BSDL Schema, this simply requires changing the maxOccurs value on the chroma blocks.

6. CONCLUSION

The ultimate goal for RMC is to realize a fully programmable decoder specification model as outlined in section 1.1. Doing so would substantially shorten the work-flow from multimedia research to consumer, by obviating the need for a lengthy standardization process in order to ensure interoperability. Instead, new multimedia technology could be immediately deployed using RMC tools. Realizing this vision requires further work, particularly in on-the-fly reconfigurability for FPGA-based systems. However, the well defined component model of the CAL language, and the reconfigurability of an RMC bitstream described by BSDL are significant steps toward this goal.

Table 1. Extract from MPEG-4 syntax specification

VideoObjectPlane() {	Bits	Mnemonic
vop_start_code	32	bslbf
vop_coding_type	2	uimsbf
do {		
modulo_time_base	1	bslbf
} while (modulo_time_base != '0')		
marker_bit	1	bslbf
vop_time_increment	1-16	uimsbf
marker_bit	1	bslbf
vop_coded	1	bslbf
if (vop_coded == '0') {		
next_start_code()		
return()		
} //...and so on...		
}		

7. REFERENCES

[1] ITU-T, "Recommendation H.264: Advanced video coding for generic audiovisual services," 2005.

[2] ISO/IEC, "Working draft 3 of ISO/IEC 23001-4: Codec configuration representation," 2007.

[3] C. Lucarz et al., "Reconfigurable media coding: a new specification model for multimedia coders," in *Signal Processing Systems, IEEE Workshop on*, 2007.

[4] G. Kahn, "The semantics of a simple language for parallel programming," in *Proceedings of the IFIP Congress*. 1974, North-Holland Publishing Co.

[5] J.B. Dennis, "First version data flow procedure language," Tech. Memo MAC TM 61, MIT Lab. Comp. Sci., 1975.

[6] J. Eker and J.W. Janneck, "CAL Language Report," Tech. Memo UCB/ERL M03/48, UC Berkeley, 2003.

[7] Paul Klint et al., "Toward an engineering discipline for grammarware," *ACM Trans. on Software Engineering Methodologies*, vol. 14, no. 3, pp. 331–380, 2005.

[8] Apple, "Quicktime file format," `developer.apple.com/reference/QuickTime/`, 2001.

[9] Microsoft, "AVI/RIFF file reference," `msdn2.microsoft.com/en-us/library/ms779636`.

[10] A. Eleftheriadis and D. Hong, "Flavor: a formal language for audio-visual object representation," in *Multimedia, 12th ACM intl. conf. on*, 2004, pp. 816–819.

[11] C. Timmerer et al., "Digital item adaptation - coding format independence," in *The MPEG-21 Book*, I. Burnett et al., Eds. Wiley, Chichester, UK., 2006.

[12] ISO/IEC, "14496 Coding of audio-visual objects," 2004.

[13] J. Clark, "XSL transformations (XSLT)," `www.w3.org/TR/xslt`, 1999.