MPEG Reconfigurable Video Coding

Marco Mattavelli, Jörn W. Janneck and Mickaël Raulet

Abstract The current monolithic and lengthy scheme behind the standardization and the design of new video coding standards is becoming inappropriate to satisfy the dynamism and changing needs of the video coding community. Such a scheme and specification formalism do not enable designers to exploit the clear commonalities between the different codecs, neither at the level of the specification nor at the level of the implementation. Such a problem is one of the main reasons for the typical long time interval elapsing between the time a new idea is validated until it is implemented in consumer products as part of a worldwide standard. The analysis of this problem originated a new standard initiative within the ISO/IEC MPEG committee, called Reconfigurable Video Coding (RVC). The main idea is to develop a video coding standard that overcomes many shortcomings of the current standardization and specification process by updating and progressively incrementing a modular library of components. As the name implies, flexibility and reconfigurability are new attractive features of the RVC standard. The RVC framework is based on the usage of a new actor/dataflow oriented language called CAL for the specification of the standard library and the instantiation of the RVC decoder model. CAL dataflow models expose the intrinsic concurrency of the algorithms by employing the notions of actor programming and dataflow. This chapter gives an overview of the concepts and technologies building the standard RVC framework and the non standard tools supporting the RVC model from the instantiation and simulation of the CAL model to the software and/or hardware code synthesis.

Marco Mattavelli
Microelectronic Systems Lab, EPFL, CH-1015 Lausanne, Switzerland
e-mail: marco.mattavelli@epfl.ch

Jörn W. Janneck
University of California at Berkeley, Berkeley, CA 94720, U.S.A.
e-mail: jwj@acm.org

Mickaël Raulet
IETR/INSA Rennes, F-35043, Rennes, France
e-mail: mickael.raulet@insa-rennes.fr

S.S. Bhattacharyya et al. (eds.), *Handbook of Signal Processing Systems*,
DOI 10.1007/978-1-4419-6345-1_3,

1 Introduction

A large number of successful MPEG (Motion Picture Expert Group) video coding standards has been developed since the first MPEG-1 standard in 1988. The standardization efforts in the field, besides having as first objective to guarantee the interoperability of compression systems, have also aimed at providing appropriate forms of specifications for wide and easy deployment. While video standards are becoming increasingly complex, and they take ever longer to be produced, this makes it difficult for standards bodies to produce timely specifications that address the need to the market at any given point in time. The structure of past standards has been one of a monolithic specification together with a fixed set of *profiles* that subset the functionality and capabilities of the complete standard. Similar comments apply to the reference code, which in more recent standards has become normative itself. Video devices are typically supporting a single profile of a specific standard, or a small set of profiles. They have therefore only very limited adaptivity to the video content, or to environmental factors (bandwidth availability, quality requirements).

Within the ISO/IEC MPEG committee, Reconfigurable Video Coding (RVC) [12] [4] standard is intended to address the two following issues: make standards faster to produce, and permit video devices based on those standards to exhibit more flexibility with respect to the coding technology used for the video content. The key idea is to standardize a library of video coding components, instead of an entire video decoder. The standard can then evolve flexibly by incrementally extending that library, and video devices can configure themselves to support a variety of coding algorithms by composing encoders and decoders from that library of predefined coding modules.

This chapter gives an overview of the concepts and technologies building the standard RVC framework and can complement and be complemented by Chapter 14, Chapter 2, and Chapter 5.

2 Requirements and rationale of the MPEG RVC framework

Started in 2004, the MPEG Reconfigurable Video Coding (RVC) framework [4] is a new ISO standard currently (Fig. 1) under its final stage of standardization, aiming at providing video codec specifications at the level of library components instead of monolithic algorithms. RVC solves this problem by defining two standards: a language with which a video decoder can be described (ISO/IEC23001-4 or MPEG-B pt. 4 [10]) and a library of video coding tools employed in MPEG standards (ISO/IEC23002-4 or MPEG-C pt. 4 [11]). The new concept is to be able to specify a decoder of an existing standard or a completely new configuration that may better satisfy application-specific constraints by selecting standard components from a library of standard coding algorithms. The possibility of dynamic configuration and reconfiguration of codecs also requires new methodologies and new tools for describing the new bitstream syntaxes and the parsers of such new codecs.

The essential concepts of the RVC framework (Fig. 2) are the following:

- RVC-CAL [6], a dataflow language describing the Functional Unit (FU) behavior. 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), a language describing the video codec configurations. FNL is an XML dialect that lists the FUs composing the codec, the parameterization of these FUs and the connections between the FUs. FNL

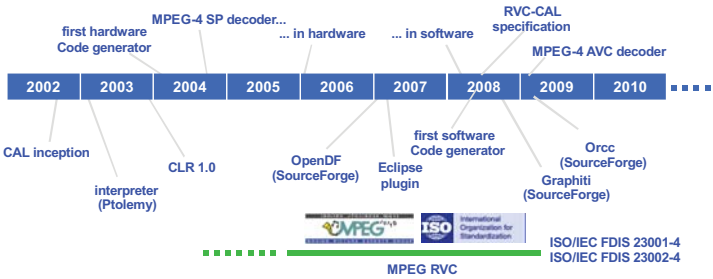

Fig. 1 CAL and RVC standard timeline.

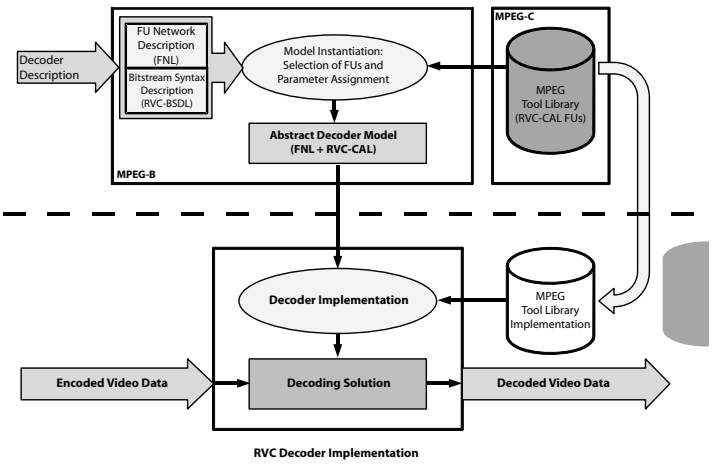

Fig. 2 RVC standard

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), a language 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 3.4.
- A library of video coding tools, 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) constituting a codec configuration (described using FNL) instantiating FUs of the MPEG Toolbox. Figure 2 depicts the process of instantiating an "Abstract Decoder Model" in RVC.
- Tools simulating and validating the behavior of the ADM (Open DataFlow environment [1]).
- Tools automatically generating software and hardware descriptions of the ADM.

2.1 Limits of previous monolithic specifications

MPEG has produced several video coding standards such as MPEG-1, MPEG-2, MPEG-4 Video, AVC (Advanced Video Coding) and recently SVC (Scalable Video Coding). While at the beginning MPEG-1 and MPEG-2 were only specified by textual descriptions, with the increasing complexity of algorithms, starting with the MPEG-4 set of standards, C or C++ specifications, called also reference software, have became the formal specification of the standards. However, the past monolithic specification of such standards (usually in the form of C/C++ programs) lacks flexibility and does not allow to use the combination of coding algorithms from different standards enabling to achieve specific design or performance trade-offs and thus fill, case by case, the requirements of specific applications. Indeed, not all coding tools defined in a *profile@level* of a specific standard are required in all application scenarios. For a given application, codecs are either not exploited at their full potential or require unnecessarily complex implementations. However, a decoder conformant to a standard has to support all of them and may results in non-efficient implementations.

Moreover, such descriptions composed of non-optimized non-modular software packages have started to show many limits. If we consider that they are in practice the starting point of any implementation, system designers have to rewrite these software packages not only to try to optimize performances, but also to transform these descriptions into appropriate forms adapted to the current system design methodologies. Such monolithic specifications hide the inherent parallelism and the dataflow structure of the video coding algorithms, features that are necessary to be exploited for efficient implementations. In the meanwhile the evolution of video coding tech-

nologies, leads to solutions that are increasingly complex to be designed and present significant overlap between successive versions of the standards.

Why C etc. Fail? The control over low-level details, which is considered a merit of C language, typically tends to over-specify programs. Not only the algorithms themselves are specified, but also how inherently parallel computations are sequenced, how and when inputs and outputs are passed between the algorithms and, at a higher level, how computations are mapped to threads, processors and application specific hardware. In general, it is not possible to recover the original knowledge about the intrinsic properties of the algorithms by means of analysis of the software program and the opportunities for restructuring transformations on imperative sequential code are very limited compared to the parallelization potential available on multi-core platforms [3]. These are the main reasons for which C has been replaced by CAL in RVC.

2.2 Reconfigurable Video Coding specification requirements

Scalable parallelism. In parallel programming, the number of things that are happening at the same time can scale in two ways: It can increase with the size of the problem or with the size of the program. Scaling a regular algorithm over larger amounts of data is a relatively well-understood problem, while building programs such that their parts execute concurrently without much interference is one of the key problems in scaling the von Neumann model. The explicit concurrency of the actor model provides a straightforward parallel composition mechanism that tends to lead to more parallelism as applications grow in size, and scheduling techniques permit scaling concurrent descriptions onto platforms with varying degrees of parallelism.

Modularity and reuse. The ability to create new abstractions by building reusable entities is a key element in every programming language. For instance, object-oriented programming has made huge contributions to the construction of von Neumann programs, and the strong encapsulation of actors along with their hierarchical composability offers an analog for parallel programs.

Concurrency. In contrast to procedural programming languages, where control flow is made explicit, the actor model emphasizes explicit specification of concurrency. Rallying around the pivotal and unifying von Neumann abstraction has resulted in a long and very successful collaboration between processor architects, compiler writers, and programmers. Yet, for many highly concurrent programs, portability has remained an elusive goal, often due to their sensitivity to timing. The untimedness and asynchrony of stream-based programming offers a solution to this problem. The portability of stream-based programs is underlined by the fact that programs of considerable complexity and size can be compiled to competitive hardware [14] as well as software [26], which suggests that stream-based programming might even be a solution to the old problem of flexibly co-synthesizing different mixes of hardware/software implementations from a single source.

Encapsulation. The success of a stream programming model will in part depend on its ability to configure dynamically and to virtualize, i.e. to map to collections of computing resources too small for the entire program at once. Moving parts of a program on and off a resource requires encapsulation, i.e. a clear distinction between those pieces that belong to the parts to be moved and those that do not. The transactional execution of actors generates points of *quiescence*, the moments between transactions, when the actor is in a defined and known state that can be safely transferred across computing resources.

3 Description of the standard or normative components of the framework

The fundamental element of the RVC framework, in the normative part, is the Decoder Description (Fig. 2) that includes two types of data:

The Bitstream Syntax Description (BSD), which describes the structure of the bitstream. The BSD is written in RVC-BSDL. It is used to generate the appropriate parser to decode the corresponding input encoded data [9] [25].

The FU Network Description (FND), which describes the connections between the coding tools (i.e. FUs). It also contains the values of the parameters used for the instantiation of the different FUs composing the decoder [5] [14] [26]. The FND is written in the so called FU Network Language (FNL). The syntax parser (built from the BSD), together with the network of FUs (built from the FND), form a CAL model called the Abstract Decoder Model (ADM), which is the normative behavioral model of the decoder.

3.1 The toolbox library

An interesting feature of the RVC standard that distinguishes it from traditional decoders-rigidly-specified video coding standards is that, a description of the decoder can be associated to the encoded data in various ways according to each application scenario. Figure 3 illustrates this conceptual view of RVC [20]. All the three types of decoders are within the RVC framework and constructed using the MPEG-B standardized languages. Hence, they all conform to the MPEG-B standard. A `Type-1` decoder is constructed using the FUs within the MPEG Video Tool Library (VTL) only. Hence, this type of decoder conforms to both the MPEG-B and MPEG-C standards. A `Type-2` decoder is constructed using FUs from the MPEG VTL as well as one or more proprietary libraries (VTL 1-n). This type of decoder conforms to the MPEG-B standard only. Finally, a `Type-3` decoder is constructed using one or more proprietary VTL (VTL 1-n), without using the MPEG VTL. This type of decoder also conforms to the MPEG-B standard only. An RVC decoder (i.e. conformant to MPEG-B) is composed of coding tools described in VTLs according

to the decoder description. The MPEG VTL is described by MPEG-C. Traditional programming paradigms (monolithic code) are not appropriate for supporting such types of modular framework. A new dataflow-based programming model is thus specified and introduced by MPEG RVC as specification formalism.

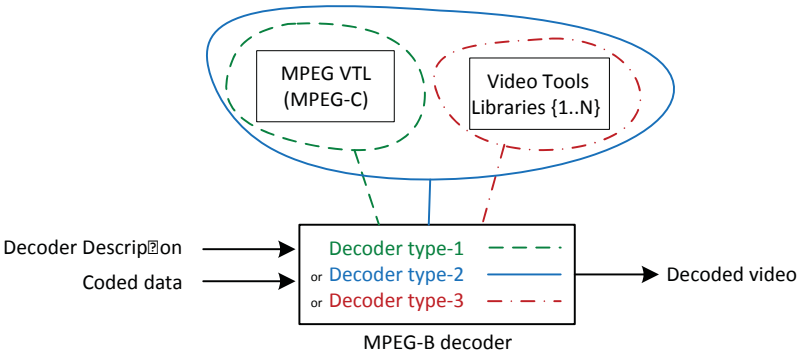

Fig. 3 The conceptual view of RVC.

The MPEG VTL is normatively specified using RVC-CAL. An appropriate level of granularity for the components of the standard library is important, to enable an effective possibility of reconfigurations, for codecs, and an efficient reuse of components in codecs implementations. If the library is composed of too coarse modules, such modules will be too large/coarse to allow their usage in different and interesting codec configurations, whereas, if the library component granularity level is too fine, the number of modules in the library will result to be too large for an efficient and practical reconfiguration process at the codec implementation side, and may obscure the desired high-level description and modeling features of the RVC codec specifications. Most of the efforts behind the standardization of the MPEG VTL were devoted to study the best granularity trade-off level of the VTL components. However, it must be noticed that the choice of the best trade-off in terms of high-level description and module re-usability, does not really affect the potential parallelism of the algorithm that can be exploited in multi-core and FPGA implementations.

3.2 The CAL *Actor Language*

CAL [6] is a domain-specific language that provides useful abstractions for dataflow programming with actors. For more information on dataflow methods, the reader may refer to Part IV of this handbook, which contains several chapters that go into detail on various kinds of dataflow techniques for design and implementation of signal processing systems. CAL has been used in a wide variety of applications and has been compiled to hardware and software implementations, and work on

mixed HW/SW implementations is under way. The next section provides a brief introduction to some key elements of the language.

3.2.1 Basic Constructs

The basic structure of a CAL actor is shown in the `Add` actor (Fig. 4), which has two input ports `A` and `B`, and one output port `Out`, all of type `T`. `T` may be of type `int`, or `uint` for respectively integers and unsigned integers, of type `bool` for booleans, or of type `float` for floating-point integers. Moreover CAL designers may assign a number of bits to the specific integer type depending on the variable numeric size. The actor contains one *action* that consumes one token on each input ports, and produces one token on the output port. An action may *fire* if the availability of tokens on the input ports matches the *port patterns*, which in this example corresponds to one token on both ports `A` and `B`.

```
actor Add() T A, T B ⇒ T Out :
  action [a], [b] ⇒ [sum]
  do
     sum := a + b;
  end
end
```

Fig. 4 Basic structure of a CAL actor.

An actor may have any number of actions. The untyped `Select` actor (Fig. 5) reads and forwards a token from either port `A` or `B`, depending on the evaluation of guard conditions. Note that each of the actions has empty bodies.

```
actor Select () S, A, B ⇒ Output:

    action S: [sel], A: [v] ⇒ [v]
    guard sel end

    action S: [sel], B: [v] ⇒ [v]
    guard not sel end
end
```

Fig. 5 Guard structure in a CAL actor.

3.2.2 Priorities and State Machines

An action may be labeled and it is possible to constrain the legal firing sequence by expressions over labels. In the `PingPongMerge` actor, reported in the Figure 6, a finite state machine *schedule* is used to force the action sequence to alternate between the two actions `A` and `B`. The schedule statement introduces two states `s1` and `s2`.

```
actor PingPongMerge () Input1, Input2 ⇒ Output:

  A: action Input1: [x] ⇒ [x] end
  B: action Input2: [x] ⇒ [x] end

  schedule fsm s1:
    s1 (A) --> s2;
    s2 (B) --> s1;
  end
end
```

Fig. 6 FSM structure in a CAL actor.

The `Route` actor, in the Figure 7, forwards the token on the input port `A` to one of the three output ports. Upon instantiation it takes two parameters, the functions `P` and `Q`, which are used as predicates in the guard conditions. The selection of which action to fire is in this example not only determined by the availability of tokens and the guards conditions, by also depends on the `priority` statement.

```
actor Route (P, Q) A ⇒ X, Y, Z:

    toX: action [v] ⇒ X: [v]
         guard P(v) end
    toY: action [v] ⇒ Y: [v]
         guard Q(v) end
    toZ: action [v] ⇒ Z: [v] end

    priority
        toX > toY > toZ;
    end
end
```

Fig. 7 Priority structure in a CAL actor.

3.2.3 CAL subset language for RVC

For an in-depth description of the language, the reader is referred to the language report [6], for the specific subset specified and standardized by ISO in the Annex C of [10]. This subset only deals with fully typed actors and some restrictions on the CAL language constructs from [6] to have efficient hardware and software code generations without changing the expressivity of the algorithm. For instance, Figures 5, 6 and 7 are not RVC-CAL compliant and must be changed as the Figures 8, 9 and 10 where *T1, T2, T* are the types and only typed paramters can be passed to the actors not functions as *P, Q*.

```
actor Select () T1 S, T2 A, T3 B ⇒ T3 Output:

    action S: [sel], A: [v] ⇒ [v]
    guard sel end

    action S: [sel], B: [v] ⇒ [v]
    guard not sel end
end
```

Fig. 8 Guard structure in a RVC-CAL actor.

```
actor PingPongMerge () T Input1, T Input2 ⇒ T Output:

  A: action Input1: [x] ⇒ [x] end
  B: action Input2: [x] ⇒ [x] end

  schedule fsm s1:
    s1 (A) --> s2;
    s2 (B) --> s1;
  end
end
```

Fig. 9 FSM structure in a RVC-CAL actor.

A large selection of example actors is available at the OpenDF repository [15], among them can also be found the MPEG-4 decoder discussed below. Many other actors written in RVC-CAL will be soon available at the standard MPEG RVC tool repository once the conformance testing process will be completed.

```
actor Route () T A ⇒ T X, T Y, T Z:
    funtion P(T v_in)--> T:
    \\ body of the function P
      P(v_in)
    end
    funtion Q(T v_in)--> T:
    \\ body of the function P
      Q(v_in)
    end

    toX: action [v] ⇒ X: [v]
         guard P(v) end
    toY: action [v] ⇒ Y: [v]
         guard Q(v) end
    toZ: action [v] ⇒ Z: [v] end

    priority
        toX > toY > toZ;
    end
end
```

Fig. 10 Priority structure in a RVC-CAL actor.

3.3 FU Network language for the codec configurations

A set of CAL actors are instantiated and connected to form a CAL application, i.e. a CAL network. Figure 11 shows a simple CAL network `Sum`, which consists of the previously defined RVC-CAL `Add` actor and the delay actor shown in Figure 12.

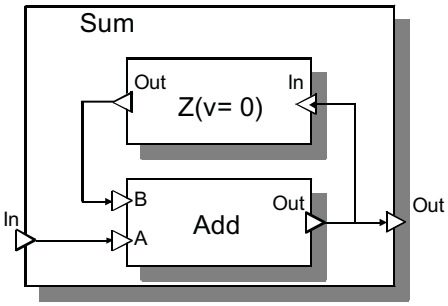

Fig. 11 A simple CAL network.

The source/language that defined the network `Sum` is found in Figure 13. Please, note that the network itself has input and output ports and that the instantiated entities may be either actors or other networks, which allow for a hierarchical design.

Formerly, networks have been traditionally described in a textual language, which can be automatically converted to FNL and vice versa — the XML dialect

```
actor Z (v) T In ⇒ T Out:

  A: action ⇒ [v] end
  B: action [x] ⇒ [x] end

  schedule fsm s0:
    s0 (A) --> s1;
    s1 (B) --> s1;
  end
end
```

Fig. 12 RVC-CAL Delay actor.

```
network Sum () In ⇒ Out:

entities
  add = Add();
  z = Z(v=0);

structure
  In --> add.A;
  z.Out --> add.B;
  add.Out --> z.In;
  add.Out --> Out;
end
```

Fig. 13 Textual representation of the Sum network.

standardized by ISO in Annex B of [10]. XML (Extensible Markup Language) is a flexible way to create common information formats. XML is a formal recommendation from the World Wide Web Consortium (W3C). XML is not a programming language, it is rather a set of rules that allow you to represent data in a structured manner. Since the rules are standard, the XML documents can be automatically generated and processed. Its use can be gauged from its name itself:

- Markup: Is a collection of Tags
- XML Tags: Identify the content of data
- Extensible: User-defined tags

The XML representation of the `Sum` network is found in Figure 14. A graphical editing framework called Graphiti editor [8] is available to create, edit, save and display a network. The XML and textual format for the network description are supported by such an editor.

```
<?xml version="1.0" encoding="UTF-8"?>
<XDF name="Sum">
   <Port kind="Input" name="In"/>
   <Port kind="Output" name="Out"/>
   <Instance id="add"/>
   <Instance id="z">
      <Class name="Z"/>
      <Parameter name="v">
         <Expr kind="Literal"
               literal-kind="Integer" value="0"/>
      </Parameter>
   </Instance>
   <Connection dst="add" dst-port="A"
         src="" src-port="In"/>
   <Connection dst="add" dst-port="B"
         src="z" src-port="Out"/>
   <Connection dst="z" dst-port="In"
         src="add" src-port="Out"/>
   <Connection dst="" dst-port="Out"
         src="add" src-port="Out"/>
</XDF>
```

Fig. 14 XML representation of the Sum network.

3.4 Bitstream syntax specification language BSDL

MPEG-B Part 5 is an ISO/IEC international standard that specifies BSDL [9] (Bitstream Syntax Description Language), an XML dialect describing generic bitstream syntaxes. In the field of video coding, the bitstream description in BSDL of MPEG-4 AVC [30] 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 15 shows a BSD associated to its corresponding BSDL schema.

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. 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 15. 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 [29]. BSDL was originally conceived and designed to enable adaptation of scalable multimedia contents in a format-independent manner [15]. 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 15 is an example BSDL description for video in MPEG-4 AVC format.

In the RVC framework, BSDL has been chosen because:

```
<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>
```

```
<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"/>
```

Fig. 15 A Bitstream Syntax Description (BSD) fragment of an MPEG-4 AVC bitstream and its corresponding BS schema fragment codec in RVC-BSDL

- it is stable and already defined by an international standard;
- the XML-based syntax interacts well with the XML-based representation of the configuration of RVC decoders;
- the parser may be easily generated from the BSDL schema by using standard tools (e.g. XSLT);
- the XML-based syntax integrates well with the XML infrastructure of the existing tools.

3.5 Instantiation of the ADM

In the RVC framework, the decoding platform acquires the Decoder Description that fully specifies the architecture of the decoder and the structure of the incoming bitstream. So as to instantiate the corresponding decoder implementation, the platform uses a library of building blocks specified by MPEG-C. Conceptually, such a library is a user defined proprietary implementation of the MPEG RVC standard library, providing the same I/O behavior. Such a library can be expressly developed to explicitly expose an additional level of concurrency and parallelism appropriate for implementing a new decoder configuration on user specific multi-core target platforms. The dataflow form of the standard RVC specification, with the associated Model of Computation, guarantee that any reconfiguration of the user defined proprietary library, developed at whatever lower level of granularity, provides an implementation that is consistent with the (abstract) RVC decoder model that is originally specified using the standard library. Figure 2 and 3 show how a decoding solution is built from, not only the standard specification of the codecs in RVC-CAL by using the normative VTL, and this already provides an explicit, concurrent and parallel model, but also from any non-normative "multi-core-friendly" proprietary Video Tool Libraries, that increases if necessary the level of explicit concurrency and parallelism for specific target platforms. Thus, the standard RVC specification, which is already an explicit model for concurrent systems, can be further improved or specialized by proprietary libraries that can be used in the instantiation phase of an RVC codec implementation.

3.6 Case study of new and existing codec configurations

3.6.1 Commonalities

All existing MPEG codecs are based on the same structure, the hybrid decoding structure including a parser that extracts values for texture reconstruction and motion compensation. Therefore, MPEG-4 SP and MPEG-4 AVC are hybrid decoders. Figure 16 shows the main functional blocks composing an hybrid decoder structure.

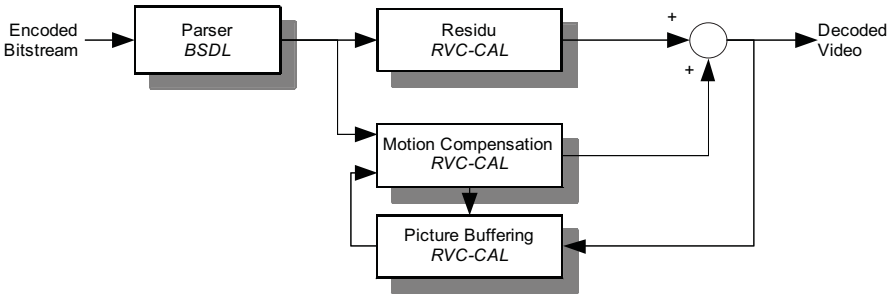

Fig. 16 Hybrid decoder structure

As said earlier, an RVC decoder is described as a block diagram with FNL [10], an XML dialect that describes the structural network of interconnected actors from the Standard MPEG Toolbox. The only 2 case studies performed so far by MPEG RVC experts [26] [14] are the RVC-CAL specifications of MPEG-4 Simple Profile decoder and MPEG-4 AVC decoder [7].

3.6.2 MPEG-4 Simple Profile (SP) Decoder

Figure 17 shows the network representation of the macroblock-based MPEG-4 Simple Profile decoder description. The parser is a hierarchical network of actors (each of them is described in a separate FNL file). All other blocks are atomic actors programmed in RVC-CAL. Figure 17 presents the structure of the MPEG-4 Simple Profile ADM as described within RVC. Essentially it is composed of four mains parts: the parser, a luminance component (Y) processing path, and two chrominance component (U, V) processing paths. Each of the path is composed by its texture decoding engine as well as its motion compensation engine (both are hierarchical RVC-CAL Functional Units).

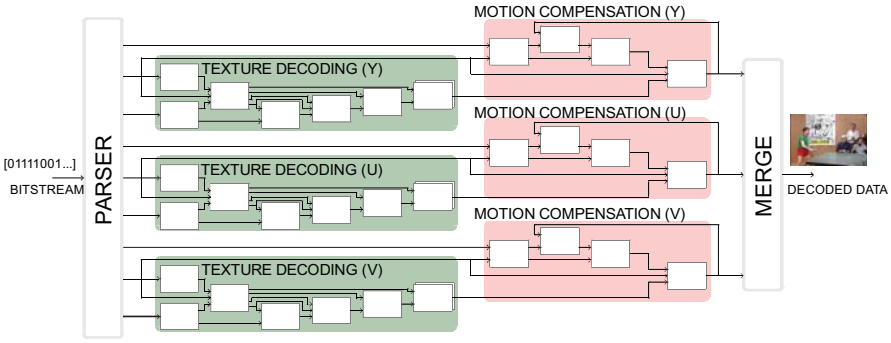

Fig. 17 MPEG-4 Simple Profile decoder description

The MPEG-4 Simple Profile abstract decoder model that essentially results to be a dataflow program (Figure 17, Table 3), is composed of 27 atomic FUs (or actors in dataflow programming) and 9 sub-networks (actor/network composition); atomic actors can be instantiated several times, for instance there are 42 actor instantiations in this dataflow program. Figure 22 shows a top-level view of the decoder. The main functional blocks include the bitstream parser, the reconstruction block, the 2D inverse cosine transform, the frame buffer and the motion compensation module. These functional units are themselves hierarchical compositions of actor networks.

3.6.3 MPEG-4 AVC Decoder

MPEG-4 Advanced Video Coding (AVC), or also know as H.264 [30], is a state-of-the-art video compression standard. Compared to previous coding standards, it is able to deliver higher video quality for a given compression ratio, and 30% better compression ratio compared to MPEG-4 SP for the same video quality. Because of its complexity, many applications including Blu-ray, iPod video, HDTV broadcasts, and various computer applications use variations of MPEG-4 AVC codec (also called *profiles*). A popular uses of MPEG-4 AVC is the encoding of high definition video contents. Due to high resolutions processing required, HD video is the application that requires the highest performance for decoding. Common formats used for HD include 720p (1280x720) and 1080p (1920x1080) resolutions, with frame rates between 24 and 60 frames per second.

The decoder introduced in this section corresponds to the *Constrained Baseline Profile* (*CBP*). This profile is primarily fitted to lowest-cost applications and corresponds to a subset of features that are in common between the *Baseline*, *Main*, and *High Profiles*.

The description of this decoder expresses the maximum of parallelism and mimics the MPEG4 SP. This description is composed of different hierarchical level. Fig. 18 shows a view of the highest hierarchy of the MPEG-4 AVC decoder — note that for readability, one input represents a group of input for similar information on each actor. The main functional block includes a parser, one *luma* and *two chroma* decoders.

The parser analyses the syntax of the bitstream with a given formal grammar. This grammar, written by hand, will later be given to the parser by a BSDL[25] description. As the execution of a parser strongly depends on the context of the bitstream, the parser incorporates a Finite State Machine so that it can sequentially extract the information from bitstream. This information passes through an entropy decoder and is then encapsulated in several kinds of tokens (residual coefficients, motion vectors...). These tokens are finally sent to the selected input port of the *luma/chroma* decoding actor.

Because decoding a *luma/chroma* component does not need to share information with the other *luma/chroma* component, we choose to encapsulate each *luma/chroma* decoding in a single actor. This means that each decoding actor can

run independently and at the same time in a separate thread. The entire decoding component actor has the same structure.

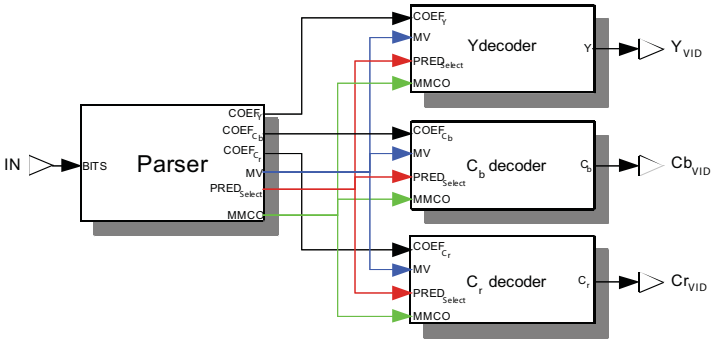

Fig. 18 Top view of MPEG-4 Advanced Video Coding decoder description.

Luma/chroma decoding actors (Fig. 19) decode a picture and store the decoded picture for later use in inter-prediction process. Each component owns the memory needed to store pictures, encapsulates into the *Decoded Picture Buffer* (*DPB*) actor. *DPB* actor also contains the *Deblocking Filter* and is a buffering solution to regulate and reorganize the resulting video flow according to the *Memory Management Control Operations* (*MMCO*) input.

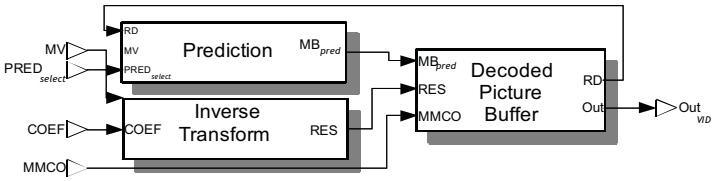

Fig. 19 Structure of decoding actors.

The *Decoded Picture Buffer* creates each frame by adding prediction data, provided by the actor *prediction*, and residual data, provided by the actor *Inverse Transform*. The *Prediction* actor (Fig. 20) encompasses*inter/intra prediction* modes and a multiplexer that sends prediction results to the output port. The $PRED_{select}$ input port has the role to stoke the right actors contingent on a prediction mode. The target of this structure is to offer a quasi-static work of the global actor and, by adding or removing prediction modes, to easily switch between configurations of the decoder. For instance, adding *B inter-prediction mode* into this structure switches the decoder into the *main profile* configuration.

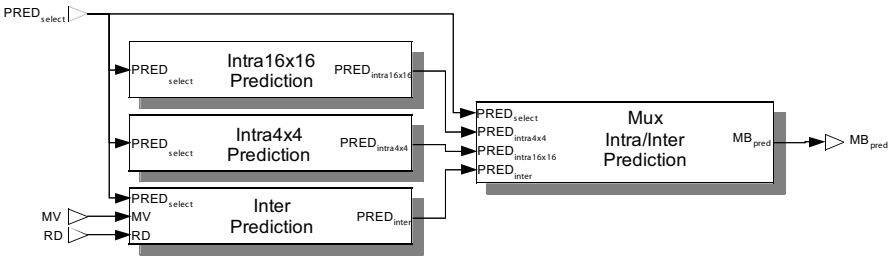

Fig. 20 Structure of prediction actor.

4 The procedures and tools supporting decoder implementations

4.1 OpenDF supporting framework

CAL is supported by a portable interpreter infrastructure that can simulate a hierarchical network of actors. This interpreter was first used in the Moses [21] project. Moses features a graphical network editor, and allows the user to monitor actors execution (actor state and token values). The project being no longer maintained, it has been superseded by an Eclipse environment composed of 2 tools/plugins—the Open Dataflow environment for CAL editing (OpenDF [15] for short) and the Graphiti editor for graphically editing the network.

One interesting and very attracting implementation methodology of MPEG RVC decoder descriptions is the direct synthesis of the standard specification. OpenDF is also a compilation framework. It provides a source of relevant application of realistic sizes and complexity and also enables meaningful experiments and advances in dataflow programming. More details on the software and hardware code generators can be found in [13] [31]. Today there exists a backend for generation of HDL (VHDL/Verilog) [14] [13]. A second backend targeting ARM11 and embedded C is under development [22] as part of the EU project ACTORS [2]. It is also possible to simulate CAL models in the Ptolemy II [4] environment.

Works made on action synthesis and actor synthesis [26] [31] led to the creation of a new compiler framework called Open RVC CAL Compiler [27]. This framework is designed to support multiple language front-ends, each of which translates actors written in RVC-CAL and FNL network into an Intermediate Representation (IR), and to support multiple language back-ends, each of which translates the Intermediate Representation into the supported languages. IR provides a dataflow representation that can be easily transformed in low level languages. Currently the only available backend is a C language backend.

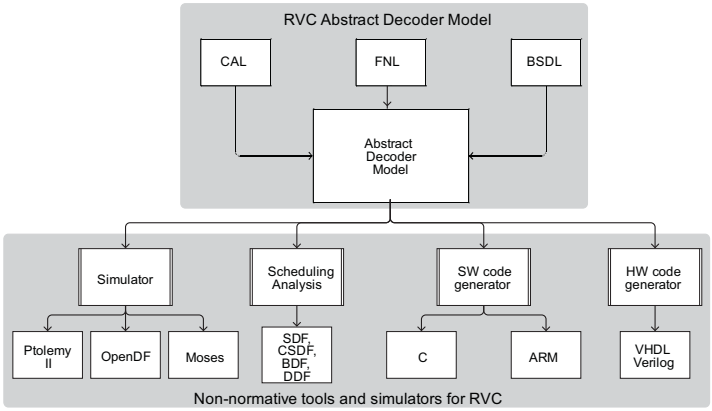

Fig. 21 OpenDF: tools

4.2 CAL2HDL synthesis

Some of the authors have performed an implementation study [13], in which the RVC MPEG-4 Simple Profile decoder specified in CAL according to the MPEG RVC formalism has been implemented on an FPGA using a CAL-to-RTL code generator called Cal2HDL. The objective of the design was to support 30 frames of 1080p in the YUV420 format per second, which amounts to a production of 93.3 Mbyte of video output per second. The given target clock rate of 120 MHz implies 1.29 cycles of processing per output sample on average.

The results of the implementation study were encouraging in that the code generated from the MPEG RVC CAL specification did not only outperform the handwritten reference in VHDL, both in terms of throughput and silicon area, but also allowed for a significantly reduced development effort. Table 1 shows the comparison between CAL specification and the VHDL reference implemented over a Xilinx Virtex 2 pro FPGA running at 100MHz.

It should be emphasized that this counter-intuitive result cannot be attributed to the sophistication of the synthesis tool. On the contrary the tool does not perform a number of potential optimizations, such as for instance optimizations involving more than one actor. Instead, the good results appear to be yield by the implementation and development process itself. The implementation approach was based generating a proprietary implementation of the standard MPEG RVC toolbox composed of FUs of lower level of granularity. Thus the implementation methodology was to substitute the FU of the standard abstract decoder model of the MPEG-4 SP with an equivalent implementation, in terms of behavior. Essentially standard toolbox FUs were substituted with networks of FU described as actors of lower granularity.

The initial design cycle of the proprietary RVC library resulted in an implementation that was not only inferior to the VHDL reference, but one that also failed to meet the throughput and area constraints. Subsequent iterations explored sev-

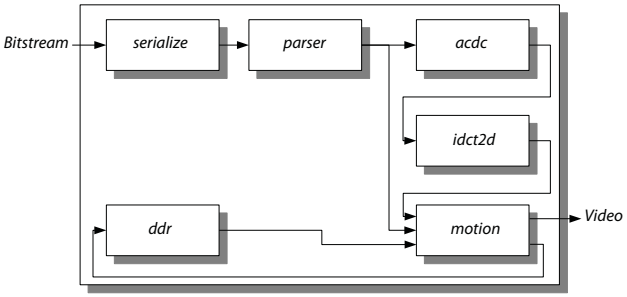

Fig. 22 Top-level dataflow graph of the proprietary implementation of the RVC MPEG-4 decoder.

eral other points in the design space until arriving at a solution that satisfied the constraints. At least for the considered implementation study, the benefit of short design cycles seem to outweigh the inefficiencies that resulted from high-level synthesis and the reduced control over implementation details.

	Size	**Speed**	**Code size**	**Dev. time**
	slices, BRAM	kMB/s	kSLOC	MM
CAL	3872, 22	290	4	3
VHDL	4637, 26	180	15	12
Improv. factor	1.2	1.6	3.75	4

kMB/s=kilo macroblocks per second
kSLOC=kilo source lines of code

Table 1 Hardware synthesis results for a proprietary implementation of a MPEG-4 Simple Profile decoder. The numbers are compared with a reference hand written design in VHDL.

In particular, the asynchrony of the programming model and its realization in hardware allowed for convenient experiments with design ideas. Local changes, involving only one or a few actors, do not break the rest of the system in spite of a significantly modified temporal behavior. In contrast, any design methodology that relies on precise specification of timing —such as RTL, where designers specify behavior cycle-by-cycle— would have resulted in changes that propagate through the design.

Table 1 shows the quality of result produced by the RTL synthesis engine of the MPEG-4 Simple Profile video decoder. Note that the code generated from the high-level dataflow RVC description and proprietary implementation of the MPEG toolbox actually outperforms the hand-written VHDL design in terms of both throughput and silicon area for a FPGA implementation.

4.3 CAL2C synthesis

Another synthesis tool called Cal2C [26] [31] currently available at [27] validates another implementation methodology of the MPEG-4 Simple Profile dataflow program provided by the RVC standard (Figure 17). The SW code generator presented in details in [26] uses process network model of computation [16] to implement the CAL dataflow model. The compiler creates a multi-thread program from the given dataflow model, where each actor is translated into a thread and the connectivity between actors is implemented via software FIFOs. Although the generation provides correct SW implementations, inherent context switches occur during execution, due to the concurrent execution of threads, which may lead to inefficient SW execution if the granularity of actor is too fine.

Major problems with multi-threaded programs are discussed in [18]. A more appropriate solution that avoids thread management are presented in [19] [23]. Instead of suspending and resuming threads based on the blocking read semantic of process network [17], actors are, instead, managed by a user-level scheduler that select the sequence of actor firing. The scheduler checks, before executing an actor, if it can fire, depending on the availability of tokens on inputs and the availability of rooms on outputs. If the actor can fire, it is executed (these two steps refers to the *enabling function* and the *invoking function* of [23]). If the actor cannot fire, the scheduler simply tests the next actor to fire (sorted following an appropriate given strategy) and so on. This code generator based on this concept [31] is available at [27]. Such a compiler presents a scheduler that has the two following characteristics: (1) actor firings are checked at run-time (the dataflow model is not scheduled statically), (2) the scheduler executes actors following a round-robin strategy (actors are sorted a priori).

In the case of the standard RVC MPEG-4 SP dataflow model such a generated mono-thread implementation is about 4 times faster than the one obtainable by [26]. Table 2 shows that synthesized C-software is faster than the simulated CAL dataflow program (80 frames/s instead of 0.15 frames/s), and twice the real-time decoding for a QCIF format (25 frames/s). However it remains slower than the automatically synthesized hardware description by Cal2HDL [13].

MPEG4 SP decoder	**Speed** kMB/s	**Clock speed** GHz	**Code size** kSLOC
CAL simulator	0.015	2.5	3.4
Cal2C	8	2.5	10.4
Cal2HDL	290	0.12	4

Table 2 MPEG-4 Simple Profile decoder speed and SLOC.

As described above, the MPEG-4 Simple Profile dataflow program is composed of 61 actor instantiations in the flattened dataflow program. The flattened network becomes a C file that currently contains a round robin scheduler for the actor scheduling and FIFOs connections between actors. Each actor becomes a C file con-

taining all its action/processing with its overall action scheduling/control. Its number of SLOC is shown in Table 3. All of the generated files are successfully compiled by gcc. For instance, the "ParserHeader" actor inside the "Parser" network is the most complex actor with multiple actions. The translated C-file (with actions and state variables) includes 2062 SLOC for both actions and action scheduling. The original CAL file contains 962 lines of codes as a comparison.

MPEG-4 SP decoder	CAL	C actors	C scheduler
Number of files	27	61	1
Code Size (kSLOC)	2.9	19	2

Table 3 Code size and number of files automatically generated for MPEG-4 Simple Profile decoder.

A comparison of the CAL description (Tab. 4) shows that the MPEG-4 AVC CAL decoder is twice more complex in RVC-CAL than the MPEG-4 Simple Profile CAL description. Some parts of the model have already been redesign in order to improve pipelining and parallelism between actors. A simulation of the MPEG-4 AVC CAL model on a *Intel Core 2 Duo* @ 2.5Ghz is more than 2.5 slower than the RVC MPEG-4 Simple Profile description.

Comparing to the MPEG-4 Simple Profile CAL model, the MPEG-4 AVC decoder has been modeled to use more CAL possibility (for instance processing of several tokens in one firing) while staying fully RVC conformant. Thanks to this increasing complexity, MPEG-4 AVC CAL model is the most reliable way to test the accordance and the efficiency of the current RVC tools. The current SW code generation of MPEG-4 AVC is promising since we can achieve up to 53 fps.

MPEG-4 AVC decoder	CAL	C actors	C scheduler
Number of files	43	83	1
Code Size (kSLOC)	5.8	44	0.9

Table 4 Code size and number of files automatically generated for MPEG-4 AVC decoder.

5 Conclusion

This chapter describes the essential components of the ISO/IEC MPEG Reconfigurable Video Coding framework based on the dataflow concept. The RVC MPEG tool library, that covers in modular form all video algorithms from the different MPEG video coding standards, shows that dataflow programming is an appropriate way to build complex heterogeneous systems from high level system specifications. The MPEG RVC framework is also supported by a simulator, software and hardware code synthesis. CAL dataflow models used by the MPEG RVC standard result also

particularly efficient for specifying signal processing systems in a very synthetic form compared to classical imperative languages. Moreover, CAL model libraries can be developed in the form of libraries of proprietary implementations of standard RVC components to describe architectural features of the desired implementation platform, thus enabling the RVC implementer/designer to work at level of abstraction comparable to the one of the RVC video coding algorithms. Hardware and software code generators then provide the low level implementation of the actors and associated network of actors for the different target implementation platforms (multi-core processors or FPGA).

References

1. Open DataFlow Sourceforge Project. http://opendf.sourceforge.net/
2. Actors FP7 project: http://www.actors-project.eu
3. Bhattacharyya, S.S., Brebner, G., Janneck, J.W., Eker, J., von Platen, C., Mattavelli, M., Raulet, M.: OpenDF: a dataflow toolset for reconfigurable hardware and multicore systems. SIGARCH Comput. Archit. News **36**(5), 29–35 (2008). DOI 10.1145/1556444.1556449
4. Bhattacharyya, S.S., Eker, J., Janneck, J.W., Lucarz, C., Mattavelli, M., Raulet, M.: Overview of the MPEG reconfigurable video coding framework. Journal of Signal Processing Systems (2009). DOI 10.1007/s11265-009-0399-3
5. Ding, D., Yu, L., Lucarz, C., Mattavelli, M.: Video decoder reconfigurations and AVS extensions in the new MPEG reconfigurable video coding framework. In: Signal Processing Systems, 2008. SiPS 2008. IEEE Workshop on, pp. 164–169 (2008). DOI 10.1109/SIPS.2008.4671756
6. Eker, J., Janneck, J.W.: CAL Language Report Specification of the CAL Actor Language. Tech. Rep. UCB/ERL M03/48, EECS Department, University of California, Berkeley (2003)
7. Gorin, J., Raulet, M., Cheng4, Y.L., Lin, H.Y., Siret, N., Sugimoto, K., Lee, G.: An RVC dataflow description of the AVC Constrained Baseline Profile decoder. In: IEEE International Conference on Image Processing, Special Session on Reconfigurable Video Coding. Cairo, Egypt (2009)
8. Graphiti Editor sourceforge: URL `http://graphiti-editor.sf.net`
9. International Standard ISO/IEC FDIS 23001-5: MPEG systems technologies - Part 5: Bitstream Syntax Description Language (BSDL)
10. ISO/IEC FDIS 23001-4: MPEG systems technologies – Part 4: Codec Configuration Representation (2009)
11. ISO/IEC FDIS 23002-4: MPEG video technologies – Part 4: Video tool library (2009)
12. Jang, E.S., Ohm, J., Mattavelli, M.: Whitepaper on Reconfigurable Video Coding (RVC). In: ISO/IEC JTC1/SC29/WG11 document N9586. Antalya, Turkey (2008). URL `http://www.chiariglione.org/mpeg/technologies/mpb-rvc/index.htm`
13. Janneck, J., Miller, I., Parlour, D., Roquier, G., Wipliez, M., Raulet, M.: Synthesizing hardware from dataflow programs. Journal of Signal Processing Systems (2009). DOI 10.1007/s11265-009-0397-5. URL `http://dx.doi.org/10.1007/s11265-009-0397-5`
14. Janneck, J.W., Miller, I.D., Parlour, D.B., Roquier, G., Wipliez, M., Raulet, M.: Synthesizing hardware from dataflow programs: An MPEG-4 simple profile decoder case study. In: Signal Processing Systems, 2008. SiPS 2008. IEEE Workshop on, pp. 287–292 (2008). DOI 10.1109/SIPS.2008.4671777
15. Joseph, A., Thomas-Kerr, I., Burnett, S., Ritz, C., Devillers, S., Schrijver, D.D., Walle, R.: Is that a fish in your ear? a universal metalanguage for multimedia. Multimedia, IEEE **14**(2), 72–77 (2007). DOI {10.1109/MMUL.2007.38}

16. Kahn, G.: The semantics of a simple language for parallel programming. In: J.L. Rosenfeld (ed.) Information processing, pp. 471–475. North Holland, Amsterdam, Stockholm, Sweden (1974)
17. Kahn, G., MacQueen, D.B.: Coroutines and networks of parallel processes. In: IFIP Congress, pp. 993–998 (1977)
18. Lee, E.A.: The problem with threads. IEEE Computer Society **39**(5), 33–42 (2006). DOI http://doi.ieeecomputersociety.org/10.1109/MC.2006.180
19. Lee, E.A., Parks, T.M.: Dataflow Process Networks. Proceedings of the IEEE **83**(5), 773–801 (1995)
20. Lucarz, C., Amer, I., Mattavelli, M.: Reconfigurable Video Coding: Concepts and Technologies. In: IEEE International Conference on Image Processing, Special Session on Reconfigurable Video Coding. Cairo, Egypt (2009)
21. Moses project: URL `http://www.tik.ee.ethz.ch/moses/`
22. von Platen, C., Eker, J.: Efficient realization of a cal video decoder on a mobile terminal (position paper). In: Signal Processing Systems, 2008. SiPS 2008. IEEE Workshop on, pp. 176–181 (2008). DOI 10.1109/SIPS.2008.4671758
23. Plishker, W., Sane, N., Kiemb, M., Anand, K., Bhattacharyya, S.S.: Functional DIF for Rapid Prototyping. In: Proceedings of the 2008 The 19th IEEE/IFIP International Symposium on Rapid System Prototyping - Volume 00, pp. 17–23. IEEE Computer Society (2008)
24. Ptolemy II: URL `http://ptolemy.eecs.berkely.edu`
25. Raulet, M., Piat, J., Lucarz, C., Mattavelli, M.: Validation of bitstream syntax and synthesis of parsers in the MPEG Reconfigurable Video Coding framework. In: Signal Processing Systems, 2008. SiPS 2008. IEEE Workshop on, pp. 293–298 (2008). DOI 10.1109/SIPS.2008.4671778
26. Roquier, G., Wipliez, M., Raulet, M., Janneck, J.W., Miller, I.D., Parlour, D.B.: Automatic software synthesis of dataflow program: An MPEG-4 simple profile decoder case study. In: Signal Processing Systems, 2008. SiPS 2008. IEEE Workshop on, pp. 281–286 (2008). DOI 10.1109/SIPS.2008.4671776
27. The Open RVC CAL Compiler project sourceforge: URL `http://orcc.sf.net`
28. The OpenDF dataflow project sourceforge: URL `http://opendf.sf.net`
29. Thomas-Kerr, J., Janneck, J., Mattavelli, M., Burnett, I., Ritz, C.: Reconfigurable media coding: Self-Describing multimedia bitstreams. In: Signal Processing Systems, 2007 IEEE Workshop on, pp. 319–324 (2007). DOI {10.1109/SIPS.2007.4387565}
30. Wiegand, T., Sullivan, G., Bjontegaard, G., Luthra, A.: Overview of the H.264/AVC video coding standard. Circuits and Systems for Video Technology, IEEE Transactions on **13**(7), 560–576 (2003). DOI {10.1109/TCSVT.2003.815165}
31. Wipliez, M., Roquier, G., Nezan, J.: Software code generation for the RVC-CAL language. Journal of Signal Processing Systems (2009). DOI 10.1007/s11265-009-0390-z. URL `http://dx.doi.org/10.1007/s11265-009-0390-z`