MPEG Reconfigurable Video Coding

Marco Mattavelli, Mickaël Raulet, and Jörn W. Janneck

Abstract Traditional efforts in standardizing video coding used to involve a lengthy process that resulted in large monolithic standards and reference codes. This approach has become increasingly ill-suited to the dynamics and the fast changing needs of the video coding community. Most importantly, there used to be no principled approach to leveraging the significant commonalities between the different codecs, neither at the level of the specification nor at the level of the implementation. The result is a long interval between the time a new idea is validated and the time it is implemented in consumer products as part of a worldwide standard. The analysis of this problem was the starting point of 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

M. Mattavelli
SCI-STI-MM Lab, EPFL, CH-1015 Lausanne, Switzerland
e-mail: marco.mattavelli@epfl.ch

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

J.W. Janneck
Department of Computer Science, Lund University, Sweden
e-mail: jwj@cs.lth.se

S.S. Bhattacharyya et al. (eds.), *Handbook of Signal Processing Systems*,
DOI 10.1007/978-1-4614-6859-2_10,

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.

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 [12]. 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) [5, 28, 43] 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 the following chapters of this handbook [11, 36, 42].

2 Requirements and Rationale of the MPEG RVC Framework

Started in 2004, the MPEG Reconfigurable Video Coding (RVC) framework [5] is a new ISO standard (Fig. 1) aiming at providing an alternative form of video codec specifications by standardizing a library of modular dataflow components instead of monolithic sequential algorithms. RVC provides the new form of specification by

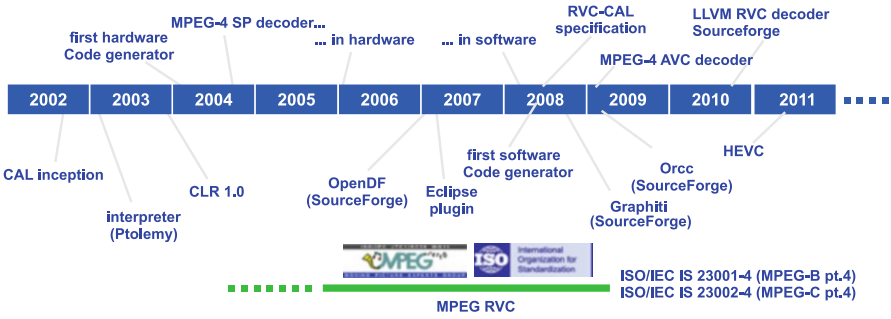


Fig. 1 CAL and RVC standard timeline

defining two standard elements: dataflow language with which video decoders can be described (ISO/IEC23001-4 or MPEG-B pt. 4 [26]) and a library of video coding tools employed in MPEG standards (ISO/IEC23002-4 or MPEG-C pt. 4 [27]). 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. Such possibility also requires new methodologies and new tools for describing the new bitstream syntaxes and the parsers of such new codecs. An additional possibility of RVC is also to allow the dynamic reconfiguration of codecs on terminal at runtime. Such new option requires normative extensions of the system layer for the transport of the new configuration and the associated signalling and is currently under study by the MPEG committee.

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

- RVC-CAL [15], a dataflow language describing the Functional Unit (FU) behavior. This language defines the behavior of dataflow components called actors (or FUs in MPEG), 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 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

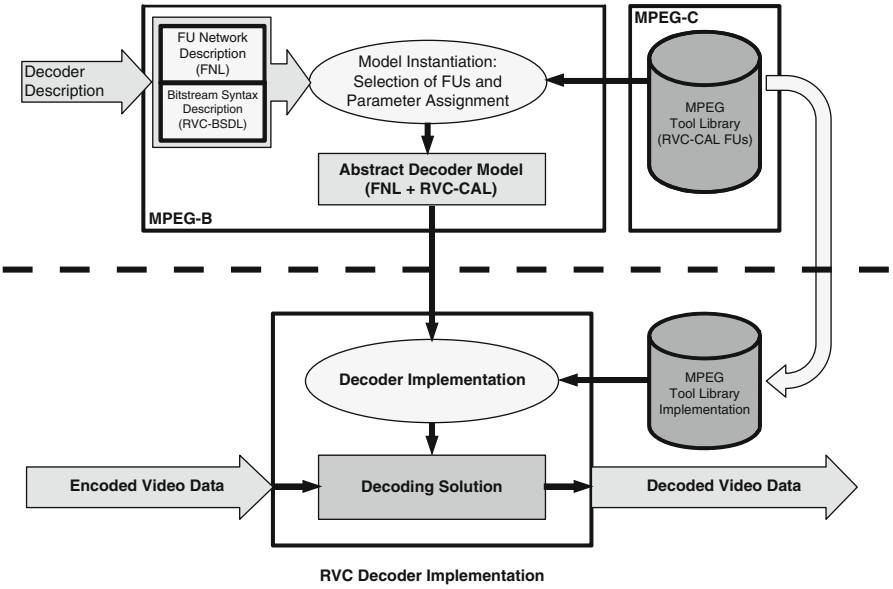


Fig. 2 RVC standard

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 Sect. 4.4.

- 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 that is standardized by MPEG) 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 [46]).
- Tools automatically generating software and hardware descriptions of the ADM.

3 Rationale for Changing the Traditional Specification Paradigm based on Sequential Model of Computation

As briefly introduced in the previous section one of the more radical innovation introduced by the RVC standard is the adoption of a non traditional model of computation and new specification language. The essential reasons of this change versus the traditional ways of building specifications are discussed here in more depth.

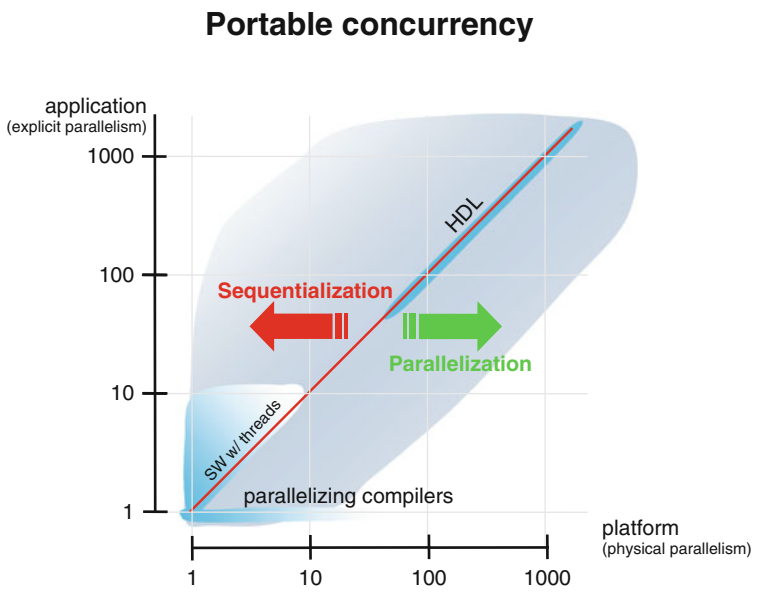


Fig. 3 Representation of a unified SW-HW design space, showing how leaving current sequential Von Neumann based approaches, portable parallelism and Software/Hardware unified programming/design can be achieved in a much larger design space by developing efficient sequentialization and parallelization techniques. Currently only design spaces labeled in the picture as "HDL" "Software w/threads" and "parallelising compilers" are covered. Dataflow approach will allow to cover a much larger space (*gray area*)

For most of the history of silicon-based computing, the relentless scaling of silicon technology has led to ever faster sequential computing machines, with higher clock rates and more sophisticated internal architectures exploiting the improvements in silicon manufacturing Fig. 3. Backwards compatible processor designs ensured that software remained portable to new machines, which meant that legacy software automatically benefited from any progress in the way processors were built. In the same way the specification of complex algorithms such as the one employed in video compression and the development of the associated reference SW descriptions have been following the same trend using the generic sequential programming languages. In recent years, however, this has ceased to be the case. In spite of continued scaling of silicon technology, individual sequential processors are not becoming faster any more, but slightly slower while reducing power dissipation. Consequently, rather than building more sophisticated and complex single processors, manufacturers have used the space gained from scaling the technology by building more processors onto a single chip, making multi-core machines and heterogeneous systems a nearly ubiquitous commodity in a wide (and increasing) range of computing applications. As a result, the performance gains of modern computing machines are primarily due to an increase in the available parallelism.

These developments pose a qualitatively novel challenge to the portability of specifications, applications and ultimately on the software that is used to implement them, as well as to the software engineering and implementation methodology in general: while sequential software used to automatically execute faster on a faster processor, an increase in performance of an application on a new platform that provides more parallelism is predicated on the ability to effectively exploit that parallelism, i.e. to parallelize the application and thus match it to the respective computing substrate. Traditionally, applications described in the style of mostly sequential algorithms have taken advantage of multiple execution units using threads and processes, thereby explicitly structuring an application into a (usually small) set of concurrently executing sequential activities that interact with each other using shared memory or other means of communication (e.g. messages, pipes, semaphores) often provided either by the operating system or some middleware. However, this parallel programming approach has some significant drawbacks. First, it poses considerable engineering challenges—large collections of communicating threads are difficult to test since errors often arise due to the timing of activities in ways that cannot be detected or reproduced easily, and the languages, environments, and tools usually provide little or no support for managing the complexities of highly parallel execution. Second, a thread-based approach scales poorly across platforms with different degrees of parallelism if the number of execution units is significantly different from the number of threads. Too few execution units mean that several threads need to be dynamically scheduled onto each of them, incurring scheduling overhead. If the number of processors exceeds the number of threads, the additional processors remain unused. The consequence is that threaded application either needs to be overengineered to using as many threads as possible, with the attendant consequences for engineering cost and performance on less parallel hardware, or they will underutilize highly parallel platforms. Either way, the requirement to construct an application with a particular degree of parallelism in mind is a severe obstacle to the portability of threaded software. In an effort to implement sequential or threaded applications on platforms that provide higher degrees of parallelism than the application itself, parallelizing compilers have been used with some success. However, the effectiveness of automatic parallelization depends highly on the application and the details of the algorithm description, and it does not scale well for larger programs. For well behaving specifications and corresponding software to scale to future parallel computing platforms as seamlessly as possible, it is necessary to describe algorithms in a way that:

1. exposes as much parallelism of the application as practical,
2. provides simple and natural abstractions that help manage the high degree of parallelism and permits principled composition of and interaction between modules,
3. makes minimal assumptions about the physical architecture of the computing machine it is implemented on,
4. is efficiently implementable on a wide range of computing substrates, including traditional sequential processors, shared-memory multicores, manycore processor arrays, and programmable logic devices, as well as combinations thereof.

This is not a trivial proposition, since it implies for instance that the current body of software will not by itself be implementable efficiently on future computers, but will have to be rewritten if it is supposed to take advantage to the parallel performance of these machines. In fact, the requirements above suggest a programming style and a tool support that is effectively the antithesis of the current approach to mapping software onto parallel platforms. Today, software tends to begin as sequential code, and parallelization, either manually or automatically, is the process of adapting the sequential algorithm and code to a parallel implementation target. Looking at the above criteria, shared-memory threads for instance fulfill the first requirement, but essentially fail on the other three. By comparison, hardware description languages such as VHDL and Verilog fulfill the first two criteria, but as they fail on the third point by assuming a particular model of (clocked) time and their implementability is essentially limited to hardware and hardware-like programmable logic (FPGAs). Needless to say CAL dataflow programming is a good candidate to be able to satisfy the above requirements and for such reasons has been selected and adopted by MPEG RVC.

3.1 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 its scalable profile SVC (Scalable Video Coding). Currently a considerable effort is aiming at providing a new standard, called High Efficiency Video Coding (HEVC), yielding a factor 2 gain versus the previous AVC performance for sequence formats ranging from HDTV resolution up to the various super HDTV formats. 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 technologies, leads to solutions that are increasingly complex to be designed and present significant overlap between successive versions of the standards.

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 [4]. These in conjunction with the previously discussed motivations, are the main reasons for which C has been replaced by CAL in RVC.

3.2 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 [32] as well as software [52], which suggests that stream-based programming [33] 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.

4 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 [25, 50].

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 [14, 32, 52]. 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.

4.1 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 4 illustrates this conceptual view of RVC [41, 43]. 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-

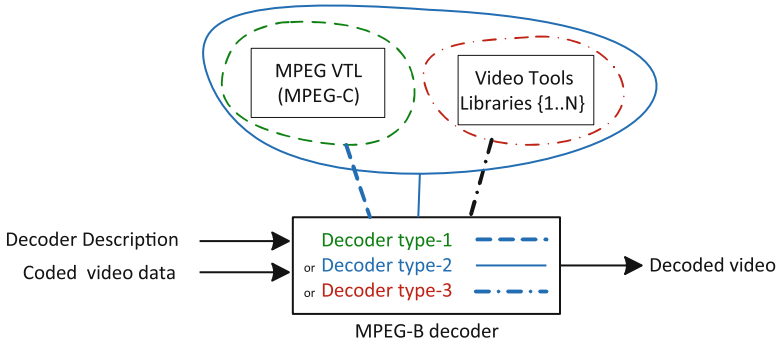


Fig. 4 The conceptual view of RVC

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.

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 granularity level of the library component 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.

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

Fig. 5 Basic structure of a CAL actor

```
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. 6 Guard structure in a CAL actor

4.2 The CAL *Actor Language*

CAL [15] is a domain-specific language that provides useful abstractions for dataflow programming with actors. 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.

4.2.1 Basic Constructs

The basic structure of a CAL actor is shown in the `Add` actor (Fig. 5), 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 numeric size of the variable. 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`.

An actor may have any number of actions. The untyped `Select` actor (Fig. 6) 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.

4.2.2 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, illustrated in Fig. 7, 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. 7 FSM structure in a CAL actor

Fig. 8 Priority structure in a CAL actor

```
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
```

The `Route` actor, in the Fig. 8, 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.

4.2.3 CAL Subset Language for RVC

For an in-depth description of the language, the reader is referred to the language report [15], for the specific subset specified and standardized by ISO in the Annex C of [26]. This subset only deals with fully typed actors and some restrictions on the CAL language constructs from [15] to have efficient hardware and software code generations without changing the expressivity of the algorithm. For instance, Figs. 6, 7 and 8 are not RVC-CAL compliant and must be changed as the Figs. 9, 10 and 11 where *T1, T2, T* are the types and only typed parameters can be passed to the actors not functions as *P, Q*.

A large selection of example actors is available at the OpenDF repository [46], among them can also be found the MPEG-4 decoder discussed below. Many other actors written in RVC-CAL are available as reference SW of the standard

```
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. 9 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. 10 FSM structure in a RVC-CAL actor

Fig. 11 Priority structure in a RVC-CAL actor

```
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
```

MPEG RVC tool repository (ISO/IEC 23002-4). Currently beside the MPEG-4 SP, MPEG-4 Part 10 AVC is available as Constrained Baseline Profile, Progressive High Profile and their scalable profile version. In parallel to the standardization process of HEVC, an RVC-CAL version of HEVC is also under development.

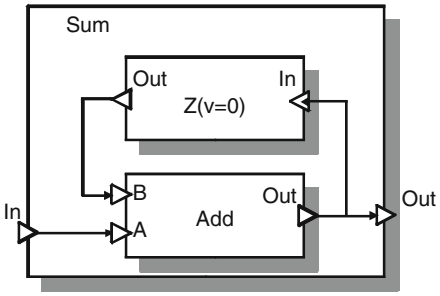


Fig. 12 A simple CAL network

```
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. 13 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. 14 Textual representation of the Sum network

4.3 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 12 shows a simple CAL network `Sum`, which consists of the previously defined RVC-CAL `Add` actor and the delay actor shown in Fig. 13.

The source/language that defined the network `Sum` is found in Fig. 14. 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 standardized by ISO in Annex B of [26]. XML (Extensible Markup Language) is a

Fig. 15 XML representation of the Sum network

```
<?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>
```

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 Fig. 15. A graphical editing framework called Graphiti editor [24] is available to create, edit, save and display a network. This editor supports XML and textual format for the network description.

4.4 Bitstream Syntax Specification Language BSDL

MPEG-B Part 5 is an ISO/IEC international standard that specifies BSDL [25] (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 [55] 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 16 shows a BSD associated to its corresponding BSDL schema.

```
<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. 16 A Bitstream Syntax Description (BSD) fragment of an MPEG-4 AVC bitstream and its corresponding BS schema fragment codec in RVC-BSDL

An encoded video bitstream is described as a sequence of binary elements of syntax of different lengths: some elements contain a single bit, while others contain many bits. 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 Fig. 16. 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 [53]. BSDL was originally conceived and designed to enable adaptation of scalable multimedia contents in a format-independent manner [54]. 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 16 is an example BSDL description for video in MPEG-4 AVC format.

In the RVC framework, BSDL has been chosen because:

- 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.

4.5 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 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 consistent with the (abstract) RVC decoder model—originally specified using the standard library. Figures 2 and 4 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.

4.6 Case Study of New and Existing Codec Configurations

4.6.1 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

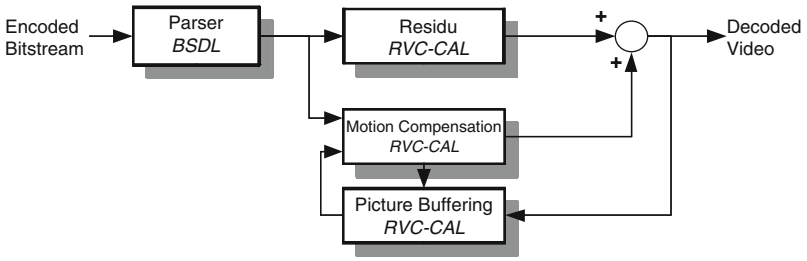


Fig. 17 Hybrid decoder structure

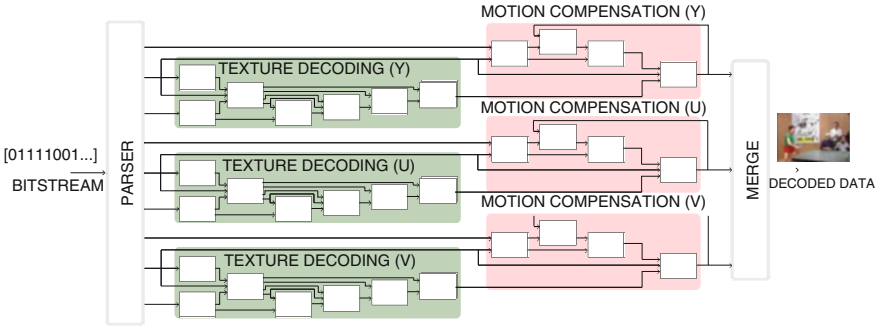


Fig. 18 MPEG-4 simple profile decoder description

motion compensation. Therefore, MPEG-4 SP and MPEG-4 AVC are hybrid decoders. Figure 17 shows the main functional blocks composing an hybrid decoder structure.

As said earlier, an RVC decoder is described as a block diagram with FNL [26], an XML dialect that describes the structural network of interconnected actors from the Standard MPEG Toolbox. The only two case studies performed so far by MPEG RVC experts [32, 52] are the RVC-CAL specifications of MPEG-4 Simple Profile decoder and MPEG-4 AVC decoder [19].

4.6.2 MPEG-4 Simple Profile (SP) Decoder

Figure 18 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 18 presents the structure of the MPEG-4 Simple Profile ADM as described within RVC. Essentially it is composed of four main 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).

The MPEG-4 Simple Profile abstract decoder model that essentially results to be a dataflow program (Fig. 18, Table 2), 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 26 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.

4.6.3 MPEG-4 AVC Decoder

The MPEG-4 Advanced Video Coding (AVC), jointly developed by ISO and ITU by which is also referred to as H.264 [55], 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 use 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 (1280×720) and 1080p (1920×1080) 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 addressing low-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 MPEG-4 SP. This description is composed of different hierarchical level. Figure 19 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 [50] 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.

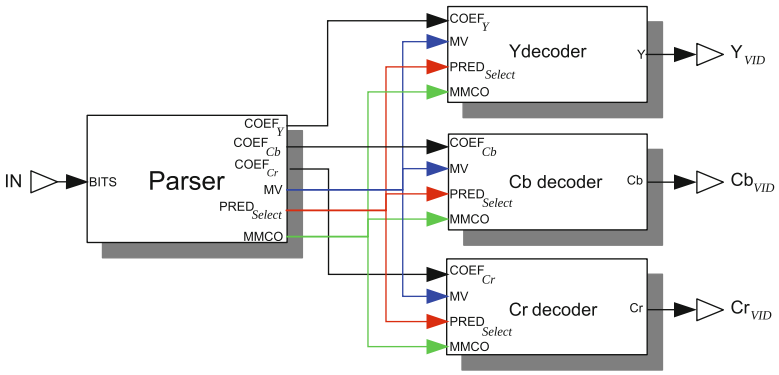


Fig. 19 Top view of MPEG-4 advanced video coding decoder description

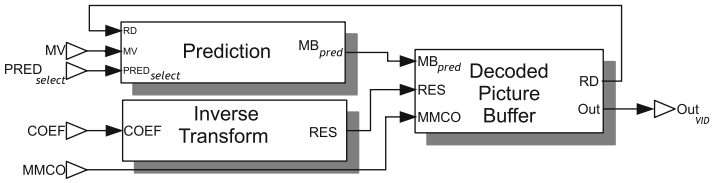


Fig. 20 Structure of decoding actors

Because decoding a *luma/chroma* component does not need to share information with the other *luma/chroma* component, it was chosen 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.

Luma/chroma decoding actors (Fig. 20) 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.

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. 21) 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.

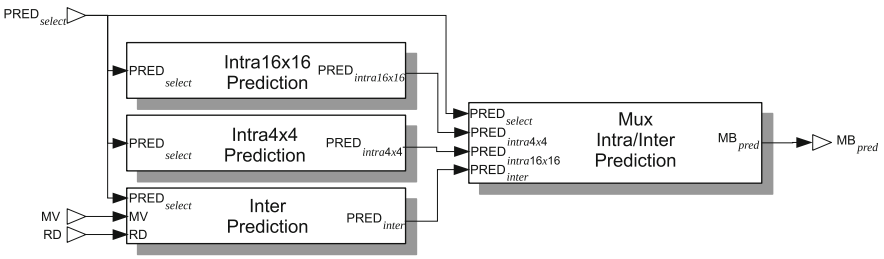


Fig. 21 Structure of prediction actor

5 The Procedures and Tools Supporting Decoder Implementations

5.1 Simulation and Compiling Frameworks

CAL is supported by portable interpreters infrastructure that can simulate a hierarchical network of actors. A first interpreter was used in the Moses [44] 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 [46] 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 [31, 56]. Today there exists a backend for generation of HDL (VHDL/Verilog) [31,32]. A second backend targeting ARM11 and embedded C has been developed [47] by the EU project ACTORS [2]. It is also possible to simulate CAL models in the Ptolemy II [49] environment.

Works made on action synthesis and actor synthesis [52,56] led to the creation of a new compiler framework called Open RVC CAL Compiler [45]. 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 several backends are available, among them a C language backend, a Java backend, a VHDL/verilog backend (using the same library as in [31,32] and a LLVM backend (Fig. 22).

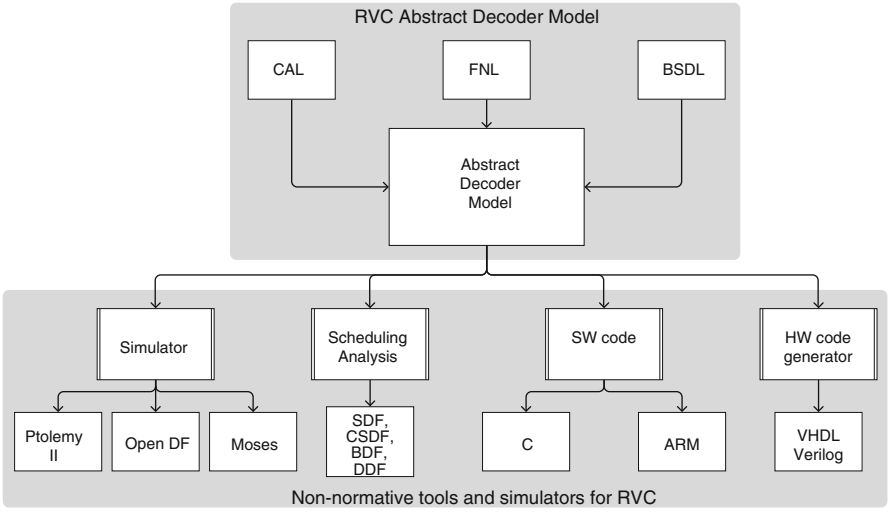
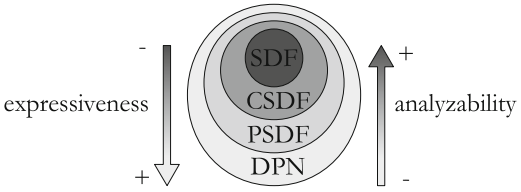


Fig. 22 Illustration of the RVC-CAL simulation and compiling tools

Fig. 23 Dataflow Models of Computation



5.2 CAL Analysis

This section presents a taxonomy of Models of Computation (MoCs) (Fig. 23) that can model the different types of behavior that RVC-CAL actors can exhibit. The actors in the Video Tool Library of the RVC standard can behave statically, cyclo-statically, quasi-statically, or dynamically. Different MoCs exist in the literature that are suitable to model these types of behaviors.

Dataflow MoCs are defined as subsets of a general model called Dataflow Process Networks (DPNs). The taxonomy shown on Fig. 23 reflects the fact that MoCs are progressively restricted from DPN towards SDF with respect to expressiveness, but at the same time they become more amenable to analysis. We first study the rules of DPN, and then present the models that can be used to model static, cyclo-static, quasi-static, and dynamic RVC-CAL actors.

Dataflow models respect the semantics of DPNs: A dataflow model is a directed graph whose vertices are *actors* and edges are unidirectional FIFO channels with unbounded capacity, connected between ports of actors.

- Each FIFO channel carries a sequence of tokens.
- Executing a DPN boils down to executing repeatedly the actors in the graph, possibly *ad infinitum*.
- An actor executes (or *fires*) when at least one of its *firing rules* is satisfied. Each firing may consume and produce tokens.
- An actor can have N firing rules, where each one represents an acceptable sequence of tokens.
- Additionally an actor has a firing function that takes a sequence of tokens and produce a sequence of tokens.

Synchronous Dataflow (SDF) [38] is the least expressive DPN model, but it is also the model that can be analyzed more easily. Schedulability and memory consumption of SDF graphs can be determined at compile-time, and algorithms exist that can map and schedule SDF graphs onto multi-processors in linear time with respect to the number of vertices and processors. Any two firing rules of an SDF actor must consume the same amount of tokens, and all firings must produce the same amount of tokens on the output ports. This definition is actually included in Lee's denotational semantics for SDF [40], which states that SDF actors have a single firing rule. Our definition simply allows SDF actors to have several firing rules as long as they have the same production/consumption rate, which in practice makes it easier to describe SDF actors that have data-dependent computations. Figure 5 represents one SDF actor in RVC-CAL.

Cyclo-static Dataflow (CSDF) [7] extends SDF with the notion of *state* while retaining the same compile-time properties concerning scheduling and memory consumption (Fig. 10). State can be represented as an *additional argument* to the firing rules and firing function, in other words it is modeled as a self-loop.

Figure 24 describes the RVC-CAL version of actor `Clip`. The $sflag$ variable is not taken into account in the firing rules, hence it does not influence the Model of Computation. Instead the *count* variable holds the number of values processed so far. Figure 25 shows the behavior of the actor `Clip` actor expressed with Cyclo-Static Dataflow (CSDF) [7]. Within this model, the number of tokens produced and consumed by an actor changes periodically according to a production/consumption sequence. For instance, the consumption sequence on port *IN* has a zero followed by 64 ones: On its first invocation, the actor will not consume any token on the port, and for the next 64 invocations, it will consume one token on the port each time. The advantage of expressing an actor as CSDF is that it can be statically scheduled because the number of tokens the actor will consume and produce is known at compile-time.

Synchronous and cyclo-static dataflow allow signal processing algorithms to be modeled as graphs with fixed production/consumption rates. On the other hand, so-called "quasi-static" graphs can be used to describe data-dependent token production and consumption. Quasi-static dataflow differs from dynamic dataflow in that there are techniques that statically schedule as many operations as possible so that only data-dependent operations are scheduled at runtime [6, 8] (see Fig. 9).

```
actor Clip ()
  int(size=10) I, bool SIGNED ⇒ int(size=9) O :

  int(size=7) count := -1;
  bool sflag;

  read_signed: action SIGNED:[s] ⇒
  guard count < 0
  do
    sflag := s;
    count := 63;
  end

  limit: action I:[i] ⇒ O:[ if i > 255 then 255
    else if i < min then min else i end
    end ]
  guard count >= 0
  var
    int min = if sflag then -255 else 0 end
  do
    count := count - 1;
  end

end
```

Fig. 24 The `Clip` actor in RVC-CAL

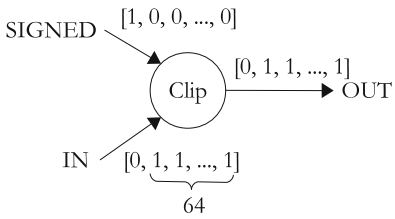


Fig. 25 Behavior of the actor `Clip` actor

RVC-CAL places no restrictions whatsoever about the firing rules nor firing function of an actor. An RVC-CAL actor can thus have a behavior that is data-independent and state-independent, cyclo-static state-dependent, quasi-static data-dependent, or data-dependent and state-dependent (dynamic). The RVC-CAL language extends the DPN MoC by adding a notion of *guard* to firing rules. Formally the guards of a firing rule are boolean predicates that may depend on the input patterns, the actor state, or both, and must be *true* for a firing rule to be satisfied.

Zebelein et al. present a classification algorithm for dynamic dataflow models in [58]. In their model, actors are defined as SystemC modules that receive and send data via SystemC FIFOs. Their classification method is based on the analysis of read and write patterns and FSMs of the different modules. Their approach is limited by the fact that they ignore any C++ code that does not contain a read or a write, and that they do not classify quasi-static actors.

In [57], Wipliez et al. present a method to automatically classify dynamic dataflow actors into more restricted dataflow MoCs, along with a method to automatically transform classified actors to static dataflow and parameterized static dataflow graphs. The transformations presented allow more efficient code to be generated for those actors and improve execution speed by reducing the number of FIFO accesses.

In [8], Boutellier et al. show how to express quasi-static RVC-CAL actors as PSDF graphs and how to derive a multiprocessor schedule from these graphs. However, they do not address the issues of automated classification and transformation: Quasi-static behavior is specified with parameters defined *manually*, and they do not explain how low-level Homogeneous SDF (HSDF) graphs created from quasi-static branches can be automatically transformed to high-level PSDF graphs. As a consequence, Wipliez and Raulet [57] can serve as a preprocessing step for their approach by automatically classifying actors as quasi-static and transforming them to high-level PSDF graphs.

Gu et al. present a technique to recognize a set of Statically Schedulable Regions (SSRs) within a dynamic dataflow program [23]. SSRs are sets of ports that are *statically coupled*, which essentially means that the production of an output port matches the consumption of the input port(s) it is connected to (additional criteria are developed in [23]). On the one hand, SSR classification has potentially more knowledge about static behavior because it looks at connected actors rather than just inside actors. On the other hand, by considering an actor as a whole our classification can discover its behavior (cyclo-static and quasi-static) and transform it into a high-level SDF or PSDF graph that will make merging easier. SSRs can be used as an input to the classification algorithm in [57] to obtain additional information.

Other approaches aiming at reducing the number of condition testing for executing actors actions are based on abstract analysis of the actor composition state space. In [29, 30] Janneck proposes a simple machine model for dataflow actors that captures the structure and the logic of selecting and executing the actions comprising an actor. The model is expected to be used to simplify and optimize the selection process based on analysis of the actor machine itself and of the actor description. In other words the actor machine model intends to capture the dynamics of actor compositions and aims at the elimination of all testing of internal buffers and action execution, thus in principle enabling the generation of very efficient code for arbitrary (i.e. dynamic) actor networks, alleviating the tension between expressiveness and efficient implementability dataflow networks.

On the same line of research, but based on the abstract exploration of actor compositions state space is the approach presented in [16, 17]. In these works a model checking methodology is used to identify the occurrence of association of recurrent states and input data token vectors, so that the identified sequences of actions executions are then used for the generation of software that omits test executions between such identified state space trajectories.

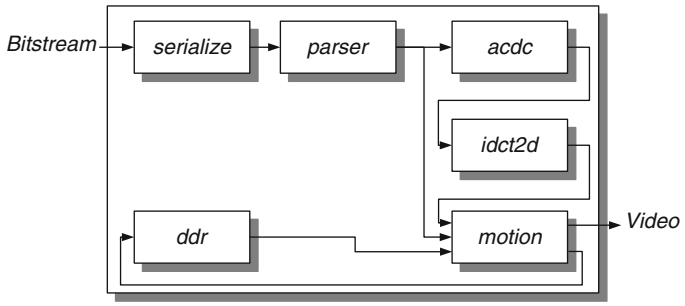


Fig. 26 Top-level dataflow graph of the proprietary implementation of the RVC MPEG-4 decoder

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

	Size slices, BRAM	Speed kMB/s	Code size kSLOC	Dev. time 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
MM= man months

5.3 CAL2HDL Synthesis

Some of the authors have performed an implementation study [31], 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 (Fig. 26).

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 100 MHz.

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 several 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.

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.

5.4 Cal2C Synthesis

Another synthesis tool called Cal2C [52, 56] currently available at [45] validates another implementation methodology of the MPEG-4 Simple Profile dataflow program (Fig. 18) provided by the RVC standard. The SW code generator presented in details in [52] uses process network model of computation [34] to implement the CAL dataflow model. The compiler creates a multi-thread program [59] 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 [39]. A more appropriate solution that avoids thread management are presented in [40, 48]. Instead of suspending and resuming threads based on the blocking read semantic of process network [35], actors are, instead, managed by a user-level scheduler that

Table 2 Code size and number of files automatically generated for MPEG-4 Simple Profile decoder

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 AVC decoder

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

select the sequence of actor firing [59]. 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 [48]). 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 [56] is available at [45]. 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).

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 containing all its action/processing with its overall action scheduling/control. Its number of SLOC is shown in Table 2. 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.

A comparison of the CAL description (Table 3) 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 redesigned in order to improve pipelining and parallelism between actors.

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 250fps on 1 core on a QCIF sequence and up to 450 fps on 2 cores.

The software code generator is also used in [9] where Boutellier et al. present a design flow that enables automatic synthesis of RVC-CAL actor networks to application specific transport-triggered architecture (TTA [13, 18]) processor networks. The functionality of the toolchain is demonstrated by applying it to an

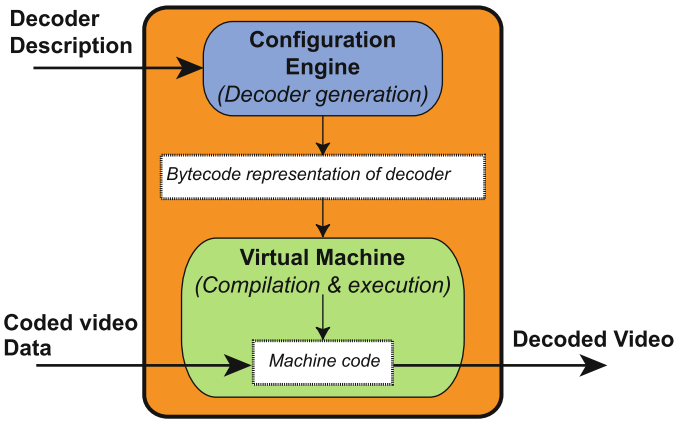


Fig. 27 Structure of the just-in-time adaptive decoder engine

RVC-CAL network, which defines an MPEG-4 SP video decoder. The automated process realizes the RVC-CAL dataflow network into several tiny heterogeneous processors.

5.5 LLVM Synthesis: An LLVM Based RVC Decoder

As shown on Fig. 4, an RVC decoder can create and handle several decoder descriptions on-the-fly, either by using coding tools standardized in MPEG, or proprietary implementations of coding tools, or other hybrid versions composed from proprietary and standardized implementations.

An RVC decoder called Jade [21, 22], represented in Fig. 27, extends a Virtual Machine to handle a RVC decoder description. Its configuration engine (Fig. 28) has two inputs, a decoder configuration and a representation of the Video Tools Library (VTL) [20] standardized in MPEG-C pt.4. It outputs a complete dataflow representation of the decoder as a set of interconnected functional processing units in byte code format. This decoder representation can then be compiled or interpreted by a specific Virtual Machine (VM). Jade is based on the open source LLVM infrastructure [37]. This VM provides efficient Just-In-Time compilation and multicore execution for a wide range of platforms (X86, X86-64, PPC, ARM, etc.).

The configuration engine of Jade contains several mechanisms to switch between different decoder representations during the decoding process. The dataflow representation of the coding tools provided by MPEG RVC gives the ability to incrementally and partially re-program a decoder when receiving new configurations from a bitstream.

The configuration and the reconfiguration of decoder are illustrated in Table 4 on the 2 representations of decoders; the *Simple Profile* (SP) from the MPEG-4

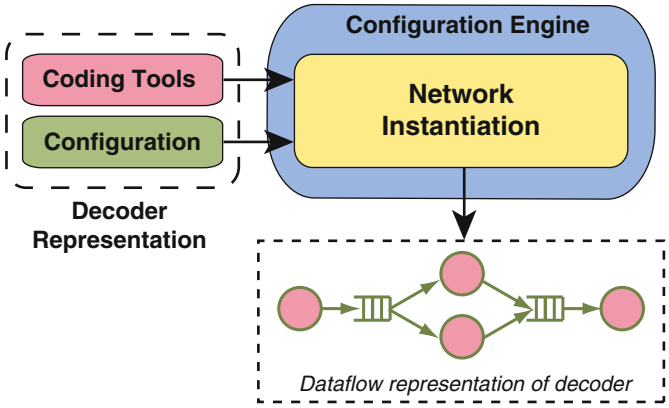


Fig. 28 Configuration of an RVC description by the configuration engine

Table 4 Configuration and reconfiguration times between implementations of MPEG-4 part 2 *Simple Profile* (SP) and MPEG-4 AVC

Standard	Configurations	Configuration	Reconfiguration
SP	RVC $->$ SP modified	1,141 ms	380 ms
AVC	CBP $->$ FRExt	3,313 ms	1,610 ms

part 2 standards and the *Constrained Baseline Profile* (CBP) of the MPEG-4 AVC. Reconfiguration is done by switching to a proprietary configuration of these decoders. The first configuration is an optimized configuration of the same decoder. The second reconfiguration is a configuration which represents the *Fidelity Range Extensions* (FRExt) of MPEG-4 AVC. The benchmarks are realized on dual-core processor at 2.4 Ghz.

Jade maximizes the use of the computing resources of any target platform by taking advantage of the inherent parallelism present in an MPEG RVC decoder. The *configuration* of a decoder gives information about the interconnection between coding tools (algorithms) that compose a decoder without carrying any implementation details for a specific platform. Therefore, Jade can execute the adaptive decoder according to the features (e.g. multiple cores) of the underlying platform.

Two optimizations algorithms based on execution models were incorporated in Jade to utilize the concurrency of a decoder configuration, depending on the number of cores in the underlying platforms. The first optimization analyzes a configuration and removes concurrency between tools to find an efficient execution on a same core. The second optimization applies an efficient distribution of independent coding tools onto separate cores.

6 Further Readings and Perspectives

This chapter introduces and describes the essential motivations and components developed so far of the ISO/IEC MPEG Reconfigurable Video Coding framework [43] based on the dataflow specification concept [4]. The MPEG RVC tool library continuously evolves so as to cover in a unified modular form all past and new standard video algorithms. MPEG RVC further demonstrates that dataflow programming is an high level system specification that constitutes an appropriate starting point and a methodology apt to build complex heterogeneous system implementations. The MPEG RVC framework is also supported by an evolving environment of different tools including simulators, software [21,22,47,52,56] and hardware [13,31] code synthesis, design exploration tools for the dimensioning of buffers, the definition of partitions and scheduling policies as well as the support of other design optimization objectives [1, 10]. CAL dataflow programs standardized and employed by MPEG RVC also result particularly efficient for developing specifications of other signal processing systems yielding very synthetic and expressive specifications when compared to classical imperative languages. In addition the strong encapsulation properties of CAL program libraries can be developed in the form of of proprietary implementation libraries enforcing the efficient usage of specific architectural features on the target implementation platform, but at the same time enabling the RVC implementer/designer to work at high levels of abstraction comparable to the one of the RVC video coding algorithm library. 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) [3, 51, 59]. In conclusion the innovations and features of both dataflow specifications and methodologies for producing the relative synthesized implementations is the subject of a very active research showing at each year new results and achievements.

References

1. A. Ab Rahman, A. Prihozhy, M. Mattavelli: Pipeline Synthesis and Optimization of FPGA-based Video Processing Applications with CAL, Eurasip Journal on Image and Video Processing, 2011, 2011:19, http://jivp.eurasipjournals.com/content/2011/1/19.
2. Actors FP7 project: http://www.actors-project.eu
3. E. Bezati, R. Thavot, G. Roquier, M. Mattavelli: High-Level Data Flow Design of Signal Processing Systems for reconfigurable and multi-core heterogeneous platforms, Journal of Real Time Image Processing, 2012.
4. S.S. Bhattacharyya, G. Brebner, J.W. Janneck, J. Eker, C. von Platen, M. Mattavelli, and M. Raulet: OpenDF: a dataflow toolset for reconfigurable hardware and multicore systems. SIGARCH Comput. Archit. News **36**(5), 29–35 (2008). DOI 10.1145/1556444.1556449
5. S.S. Bhattacharyya, J. Eker, J.W. Janneck, C. Lucarz, M. Mattavelli, and M. Raulet: Overview of the MPEG reconfigurable video coding framework. Journal of Signal Processing Systems (2011). DOI 10.1007/s11265-009-0399-3

6. B. Bhattacharya and S.S. Bhattacharyya, "Parameterized Dataflow Modeling for DSP Systems," *IEEE Transactions on Signal Processing*, vol. 49, pp. 2408–2421, 2001.
7. G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete, "Cyclo-static dataflow," *IEEE Transactions on signal processing*, vol. 44, no. 2, pp. 397–408, 1996.
8. J. Boutellier, C. Lucarz, S. Lafond, V.M. Gomez, and M. Mattavelli, "Quasi-static scheduling of CAL actor networks for reconfigurable video coding," *Journal of Signal Processing Systems*, pp. 1–12, 2009.
9. J. Boutellier, O. Silvén, and M. Raulet: "Automatic synthesis of TTA processor networks from RVC-CAL dataflow programs.," *Signal Processing Systems (SiPS), 2011 IEEE Workshop on, pp.25-30, 4-7 Oct. 2011.* DOI 10.1109/SiPS.2011.6088944
10. S. Casale Brunet, E. Bezati, R. Thavot, G. Roquier, M. Mattavelli, J. W. Janneck, Methods to Explore Design Space for MPEG RVC Codec Specifications, in Signal Processing Image Communication, Special Issue on Reconfigurable Media Coding, 2013.
11. Y. Chen and L. Chen. Video Compression. In S. S. Bhattacharyya, E. F. Deprettere, R. Leupers, and J. Takala, editors, *Handbook of Signal Processing Systems*. Springer, second edition, 2013.
12. L. Chiariglione Editor: The MPEG Representation of Digital Media. Springer Ed. 2011. DOI 10.1007/978-1-4419-6184-6_12
13. H. Corporaal: "VLIW to TTA," John Wiley & Sons, Inc., New York, NY, USA, 1997
14. D. Ding, L. Yu, C. Lucarz, and M. Mattavelli: 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
15. J. Eker and J.W. Janneck: CAL Language Report Specification of the CAL Actor Language. Tech. Rep. UCB/ERL M03/48, EECS Department, University of California, Berkeley (2003)
16. J. Ersfolk, G. Roquier, J. Lilius, M. Mattavelli Scheduling of dynamic data flow programs based on state space analysis, 2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2012, pp. 1661–1664 DOI 10.1109/ICASSP.2012.6288215.
17. J. Ersfolk, G. Roquier, F. Jokhio, J. Lilius, M. Mattavelli, Scheduling of dynamic data flow programs with model checking,, 2011 IEEE Workshop on Signal Processing Systems (SiPS), 2011, pp. 37–4 DOI 10.1109/SiPS.2011.6088946.
18. O. Esko, P. Jääskeläinen, P. Huerta, C. S. de La Lama, J. Takala, and J. I. Martinez : "Customized exposed datapath soft-core design flow with compiler support," IEEE conference on Field Programmable Logic and Applications (FPL), 2010, pp. 217–222.
19. J. Gorin, M. Raulet, Y.L. Cheng, H.Y. Lin, N. Siret, K. Sugimoto, and G. Lee: 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)
20. J. Gorin, M. Wipliez, J. Piat, M. Raulet, and F. Preteux. A portable Video Tools Library for MPEG Reconfigurable Video Coding using LLVM representation. In *Design and Architectures for Signal and Image Processing (DASIP 2010)*, pages 281–286, 2008.
21. J. Gorin, M. Wipliez, F. Preteux, and M. Raulet. LLVM-based and scalable MPEG-RVC decoder. *Journal of Real-Time Image Processing*, pages 1–12.
22. J. Gorin, M. Wipliez, M. Raulet, and F. Preteux. An LLVM-based decoder for MPEG Reconfigurable Video Coding. In *IEEE Workshop on Signal Processing Systems (SiPS 2010), Washington, D.C., USA*, pages 281–286, 2008.
23. R. Gu, J.W. Janneck, S.S. Bhattacharyya, M. Raulet, M. Wipliez, and W. Plishker, "Exploring the concurrency of an MPEG RVC decoder based on dataflow program analysis," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 19, no. 11, 2009.
24. Graphiti Editor sourceforge: URL http://graphiti-editor.sf.net
25. International Standard ISO/IEC FDIS 23001-5: MPEG systems technologies—Part 5: Bitstream Syntax Description Language (BSDL)
26. ISO/IEC International Standard 23001-4: MPEG systems technologies—Part 4: Codec Configuration Representation (2011)

27. ISO/IEC International Standard 23002-4: MPEG video technologies—Part 4: Video tool library (2010)
28. E.S. Jang, J. Ohm, and M. Mattavelli: 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
29. J. W. Janneck: Actor machines—a machine model for dataflow actors and its applications, Department of Computer Science, Lund University, Tech. Rep. LTH 96-2011, LU-CS-TR 201–247, (2011).
30. J. W. Janneck: A Machine Model for Dataflow Actors and its Applications 45th Annual Asilomar Conference on Signals, Systems, and Computers November 6–9, 2011.
31. J.W. Janneck, I.D. Miller, D.B. Parlour, G. Roquier, M. Wipliez, and M. Raulet: Synthesizing hardware from dataflow programs. Journal of Signal Processing Systems (2011). DOI 10.1007/s11265-009-0397-5
32. J.W. Janneck, I.D. Miller, D.B. Parlour, G. Roquier, M. Wipliez, and M. Raulet: 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
33. J. W. Janneck, M. Mattavelli, M. Raulet, and M. Wipliez: Reconfigurable video coding: a stream programming approach to the specification of new video coding standards. in *MMSys '10: Proceedings of the first annual ACM SIGMM conference on Multimedia systems*. USA: ACM, 2010, pp. 223–234.
34. G. Kahn: 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)
35. G. Kahn, MacQueen, D.B.: Coroutines and networks of parallel processes. In: IFIP Congress, pp. 993–998 (1977)
36. K. Konstantinides. Digital Signal Processing in Home Entertainment. In S. S. Bhattacharyya, E. F. Deprettere, R. Leupers, and J. Takala, editors, *Handbook of Signal Processing Systems*. Springer, second edition, 2013.
37. C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.
38. E.A. Lee and D.G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.
39. E.A. Lee: The problem with threads. IEEE Computer Society **39**(5), 33–42 (2006). DOI 10.1109/MC.2006.180
40. E.A. Lee and T.M. Parks: Dataflow Process Networks. Proceedings of the IEEE **83**(5), 773–801 (1995)
41. C. Lucarz, I. Amer, and M. Mattavelli: Reconfigurable Video Coding: Concepts and Technologies. In: IEEE International Conference on Image Processing, Special Session on Reconfigurable Video Coding. Cairo, Egypt (2009)
42. J. McAllister. FPGA-Based DSP. In S. S. Bhattacharyya, E. F. Deprettere, R. Leupers, and J. Takala, editors, *Handbook of Signal Processing Systems*. Springer, second edition, 2013.
43. M. Mattavelli, I. Amer, and M. Raulet, "The Reconfigurable Video Coding Standard [Standards in a Nutshell]," *Signal Processing Magazine, IEEE*, vol. 27, no. 3, pp. 159 –167, may 2010.
44. Moses project: URL http://www.tik.ee.ethz.ch/moses/
45. The Open RVC CAL Compiler project sourceforge: URL http://orcc.sf.net
46. The OpenDF dataflow project sourceforge: URL http://opendf.sf.net
47. C. von Platen and J. Eker: 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

48. W. Plishker, N. Sane, M. Kiemb, K. Anand, and S.S. Bhattacharyya: 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)
49. Ptolemy II: URL http://ptolemy.eecs.berkely.edu
50. M. Raulet, J. Piat, C. Lucarz, and M. Mattavelli: 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
51. G. Roquier, E. Bezati and M. Mattavelli: Hardware and Software Synthesis of Heterogeneous Systems from Dataflow Programs, Journal of Electrical and Computer Engineering, special issue on "ESL Design Methodology", vol. 2012, Article ID 484962, 11 pages, 2012. doi: 10.1155/2012/484962.
52. G. Roquier, M. Wipliez, M. Raulet, J.W. Janneck, I.D. Miller, and D.B. Parlour: 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
53. J. Thomas-Kerr, J.W. Janneck, M. Mattavelli, I. Burnett, and C. Ritz: 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
54. J.A. Thomas-Kerr, I. Burnett, C. Ritz, S. Devillers, D.D. Schrijver, and R. Walle: 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
55. T. Wiegand, G. Sullivan, G. Bjontegaard, and A. Luthra: 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
56. M. Wipliez, G. Roquier, and J.F. Nezan: Software code generation for the RVC-CAL language. Journal of Signal Processing Systems (2011). DOI 10.1007/s11265-009-0390-z
57. M. Wipliez and M. Raulet: Classification and transformation of dynamic dataflow programs. language. Design and Architectures for Signal and Image Processing (DASIP), 2010 Conference on, pp.303–310, 26-28 Oct. 2010. DOI 10.1109/DASIP.2010.5706280
58. C. Zebelein, J. Falk, C. Haubelt, and J. Teich, "Classification of General Data Flow Actors into Known Models of Computation," *Proc. MEMOCODE, Anaheim, CA, USA*, pp. 119–128, 2008.
59. H. Yviquel, E. Casseau, M. Wipliez, and M. Raulet: "Efficient multicore scheduling of dataflow process networks," Signal Processing Systems (SiPS), 2011 IEEE Workshop on, pp.198–203, 4-7 Oct. 2011. DOI 10.1109/SiPS.2011.6088974