MPEG Reconfigurable Video Coding: From specification to a reconfigurable implementation

Jérôme Gorin, Mickaël Raulet, Françoise Prêteux

▶ **To cite this version:**

Jérôme Gorin, Mickaël Raulet, Françoise Prêteux. MPEG Reconfigurable Video Coding: From specification to a reconfigurable implementation. Signal Processing: Image Communication, Elsevier, 2013, 28 (10), pp.1224 - 1238. <10.1016/j.image.2013.08.009>. <hal-01068867>

HAL Id: hal-01068867

https://hal.archives-ouvertes.fr/hal-01068867

Submitted on 26 Sep 2014

MPEG Reconfigurable Video Coding: from specification to a reconfigurable implementation

Jérôme Gorin[a], Mickaël Raulet[b], Françoise Prêteux[c]

[a]*OneCodec Limited, Aberdeen, UK and Boston, MA*
[b]*IETR, INSA Rennes, F-35000*
[c]*MINES ParisTech, 60 Boulevard Saint-Michel, 75272 Paris, France*

Abstract

This paper demonstrates that it is possible to produce automatic, reconfigurable, and portable implementations of multimedia decoders onto platforms with the help of the MPEG Reconfigurable Video Coding (RVC) standard. MPEG RVC is a new formalism standardized by the MPEG consortium used to specify multimedia decoders. It produces visual representations of decoder reference software, with the help of graphs that connect several coding tools from MPEG standards. The approach developed in this paper draws on Dataflow Process Networks to produce a Minimal and Canonical Representation (MCR) of MPEG RVC specifications. The MCR makes it possible to form automatic and reconfigurable implementations of decoders which can match any actual platforms. The contribution is demonstrated on one case study where a generic decoder need to process a multimedia content with the help of the RVC specification of the decoder required to process it. The overall approach is tested on two decoders from MPEG, namely MPEG-4 part 2 Simple Profile and MPEG-4 part 10 Constrained Baseline Profile. The results validate the following benefits on the MCR of decoders: compact representation, low overhead induced by its compilation, reconfiguration and multi-core abilities.

Keywords: MPEG Reconfigurable Video Coding, Dataflow Process Network, Dataflow Programming, RVC-CAL Actor Language, Video Tool Library, Open RVC-CAL Compiler, LLVM, MPEG-4 part 2 Simple Profile, MPEG-4 part 10 Advanced Video Coding

1. Introduction

For more than two decades, the **Moving Picture Experts Group** (MPEG) has provided many video coding standards such as MPEG-1, MPEG-2 and MPEG-4 and is continuing this process with the upcoming High Efficiency Video Coding (HEVC) standard. The primary goal of these standards is to assure interoperability from a provider of compressed digital multimedia content to several receivers that need to display this content. To date, video standards have consisted of textual specifications and reference software that describe a decoding process, that is the way a decoder has to interpret a video bitstream.

However, providing specifications in textual form subjects their implementation to the interpretation of the decoder designer. Their corresponding reference software, usually provided in C/C++ form, can lack of genericity, flexibility and reusability. Moreover, the potential parallelism existing between coding algorithms is entirely hidden by the sequentiality of C/C++ language. This parallelism is becoming increasingly mandatory to provide efficient implementation of decoders for hardware or multi-core platforms.

At the same time, the use of multimedia is evolving significantly. The increasing popularity of digital communication (e.g. Digital Terrestrial Television) and multimedia terminals (e.g. Smartphones) has generalized the use of multimedia standards to the largest number of customers. The multimedia market has become a highly competitive area and many players are developing their own technologies; in such a way that terminals have to support and follow these technologies so as not to become quickly in obsolescence.

To reduce interpretation problems which are damaging for standard interoperability and to accelerate adoption of new standards, MPEG designed in 2005 a new specification formalism known as MPEG *Reconfigurable Video Coding* (RVC). MPEG RVC is “a framework for developing, implementing and passing video coding technologies that favors flexibility and reuse" [1]. It aims at substituting the existing reference software written in C or HDL by abstract representations of decoders (Abstract Decoder Model or ADM). As such, MPEG RVC is based on block-diagram descriptions that make it possible to design decoders at a higher level of specification. Using this specification, a decoder is composed of a graph where vertices are coding tools and edges are communication channels between vertices. MPEG RVC aims to become a turnkey solution for providing modular, portable and user-friendly decoder descriptions. It supports multiple standards in a unified specification form, reducing time to market and facilitating terminals evolution to any future technologies.

This paper formalizes a rigorous underpinning to produce accurate, reconfigurable and efficient implementations of decoder from MPEG RVC specifications. It starts with an in depth analysis of the different aspects covered by RVC specification and match the properties of MPEG RVC with the mechanisms involved in the execution of dataflow programs. Section 2 introduces the syntax description of languages used to formalize RVC specifications. We establish in Section 3 a precise relationship between these specifications and the general concept of dataflow programming. We give a formal notation to link dataflow execution models to the paradigms from the RVC specification. Finally, in Section 4, we introduce the notion of dynamic compilation in MPEG RVC and give a concrete implementation with the development of a dedicated virtual machine able to reach dynamic implementation from specification.

2. Specification of RVC decoders

MPEG RVC has been standardized by the MPEG committee as part of two standards, namely MPEG-B *Systems Technologies* part 4 [2] and MPEG-C *Video Technologies* part 4 [3]. These two standards are continuously evolving with new amendments for upcoming or future decoder descriptions. Both standards provide specification formalism by "composing coding tools" used as a base for the standardization "by part" of MPEG decoders. As shown in Figure 1, this framework aims to produce an Abstract Decoder Model (ADM) that represents the specification of a Profile of a decoder among an existing or new MPEG standard. Technically, it is set in the form of a Video Tool Library (VTL), a Functional Unit Network Language (FNL) and a Bitstream Syntax Description Language (BSDL). All coding technologies are unified into a single library and all decoder into a single representation, as such RVC mainly focuses on reusability by allowing common coding tools to be instantiated across decoder specifications.

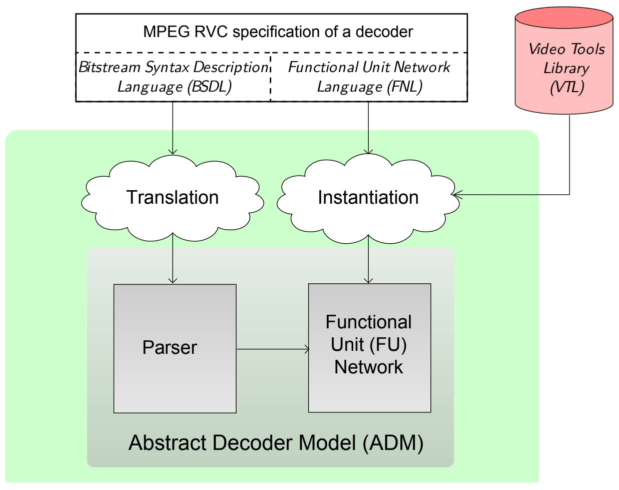
MPEG RVC specification of a decoder
Bitstream Syntax Description Language (BSDL)
Functional Unit Network Language (FNL)
Video Tools Library (VTL)
Translation
Instantiation
Parser
Functional Unit (FU) Network
Abstract Decoder Model (ADM)

Figure 1: Specification of an Abstract Decoder Model in MPEG RVC.

This section gives an overview of each of these components, except for the BSDL as we considered it outside the scope of this paper. BSDL is drawn from MPEG-B part 5 [4] and describes the parsing process of a coded bitstream for a given configuration. It is an XML document that describes each syntax element from a bitstream, which has little in common with the dataflow aspect of MPEG RVC. One research paper on the validation of BSDL within the MPEG RVC framework can be found in [5].

2.1. *MPEG-C part 4 : Video Tool Library and decoder configurations*

The MPEG-C part 4 standard [2] aims at supplying a normative standard library of video coding tools, the VTL, drawn from existing MPEG standards. It also provides an informative set of decoder configurations to form several *profiles* of MPEG decoders as networks of coding tools.

FU Name	Algo_IS_ZigzagOrAlternateHorizontalVertical_8x8	
Description	This module inverts the one-dimensional array of coefficients ordered in zigzag (*AC_PRED_DIR* = 0), alternate vertical (*AC_PRED_DIR* = 1) or alternate horizontal (*AC_PRED_DIR* = 2) scan to 2D raster order. It inputs a list of 64 integer coefficients (one per 8×8 block) and outputs the ordered list of integer according to the value of the token *AC_PRED_DIR*.	
Profiles@levels	MPEG-4 Simple Profile	
Input		
Name	Token	
AC_PRED_DIR	*AC_PRED_DIR* token	
QFS_AC	*AC* token	
Output		
Name	Token	
PQF_AC	*AC* token	
Parameter		
Name	Description	Range

Table 1: Textual description of the FU *Algo_ISZigzagOrAlternateHorizontalVertical_8x8* in *MPEG-C part 4*.

VTL represents each coding tools from MPEG standards as one Functional Unit (FU). Each FU has a textual specification that provides its purpose and a reference implementation expressed in standardized language called RVC-CAL. The textual description gives to each FU a name, a short description of its functionality, the standard and the Profile it comes from, and the properties of its input and output data. The name of FUs is normative to distinguish two kinds of use in the VTL:

1. The *algorithmic* (*ALGO*) coding tools, such as the *Inverse Discrete Cosine Transform* (IDCT), the *Inverse Quantifier* (IQ); and
2. The *data management* (*MGNT*) tools, such as data multiplexer and demultiplexer.

FUs containing algorithmic video coding tools are usually reusable between different profiles or different standards. In contrast, data management tools adapt these FUs to a specific structure and are usually specific to a decoder.

By way of example, we give in Table 1 the textual description of the FU named "Algo ISZigzagOrAlternateHorizontalVertical 8x8". It is an algorithmic coding tool, which is used in MPEG-4 part 2 to realize *inverse scan*. Inverse Scan is the process that inverse Forward Scanning at the decoder side, which assembles all non-zero coefficients in raster order for further compression.

The Inverse Scan FU comprises two inputs: *AC_PRED_DIR* and *AC_PRED_DIR* which takes data of type *QFS_AC* and *AC* respectively. The results from this computation are transmitted to one output *PQF_AC* of type *AC*. Data from ports are called *tokens* and their types refer to an identifier that details its composition.

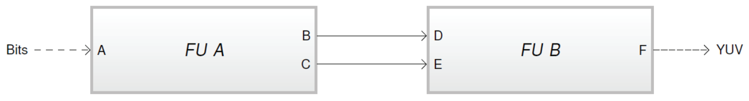

Figure 2: MPEG RVC configuration of decoders.

An Abstract Decoder Model (ADM) is configured by interconnecting coding tools from the VTL. It expresses a *configuration* to form a Profile of a decoder in a standard. This configuration corresponds to an oriented graph where vertices are the required FUs and edges are the communication dependencies between FUs. Figure 2 gives an example of a decoder configuration.

The textual description of an FU is only informative, but its identifier, ports and parameters are normative. FUs can be implemented in ADM with any software language or hardware language or can be seen as “black box” components as long as their external behavior – i.e. interfaces and data types – conforms to the description.

2.2. MPEG-B part 4 : the reference languages

The MPEG-B part 4 [2] defines the languages to form a decoder configuration: the FU Network Language, the RVC-CAL Actor Language and the BSDL. However, this standard does not impose on configurations the use of FU standardized in MPEG-C part 4. Proprietary FUs – i.e. not standardized in MPEG-C part 4 – can be added to a decoder configuration as long as they respect the MPEG-B part 4 paradigm. The resulting decoder is consequently considered to be not compliant to MPEG-C part 4 standard but only compliant with the MPEG-B part 4 standard.

The FNL expresses the network for one configuration of decoder. This network is formed as an oriented graph where vertices are *Instances* of FUs (the *Class*). An instance is defined by its identifier (*id*) and its *name* attribute. It can optionally assign values to the parameter of an actor.

```
<Input src="FU_A" src-port="A"/>
<Instance id="FU_A">
  <Class name="Algo_Example1"/>
</Instance>
<Instance id="FU_B">
  <Class name="Algo_Example2"/>
</Instance>
<Connection src="FU_A" src-port="B" dst="FU_B" dst-port="D"/>
<Connection src="FU_A" src-port="C" dst="FU_B" dst-port="E"/>
<Output src="FU_B" src-port="F"/>
```

Figure 3: A configuration of decoder using the Functional Network Language.

An FNL defines 3 types of edges: (1) between an input port of a network and an instance (*input*) (2) between an output port of an instance and an input port of another instance (*Connection*) (3) between an output port of an instance and the output port of a network (*Output*). A network can be hierarchical — this means that a network may be a part of a more general network — and can be used to pass parameters to actors. By way of example, the Figure 3 illustrates the FNL description of network in Fig. 2.

The RVC-CAL Actor Language – a subset of the CAL Actor Language [6] – gives an explicit representation of the internal computation realized by FUs. This computation is described with an entity called *actor*, which contains interfaces (input and output ports), internal states and parameters. We illustrate the actor behavior by showing in Figure 4, 5, 6 and 7 fragments of RVC-CAL specification from the FU textual description given in Table 1.

```
package org.sc29.wg11.mpeg4.part2.sp.texture;
import org.sc29.wg11.mpeg4.part2.Constants.*;

actor Algo_IS_ZigzagOrAlternateHorizontalVertical_8x8 ()
  int(size=3) AC_PRED_DIR, int(size=16) QFS_AC ==> int(size=16) PQF_AC:
```

Figure 4: Header of the RVC-CAL implementation for *Algo_ISZigzagOrAlternateHorizontalVertical_8x8*.

The *header* of an actor (Fig. 4) specifies in order the name of the actor, its interfaces and, optionally, its parameters. Each interface is strictly typed using the keyword *size*. *Packages* provide additional information to the developer on which profile and standard a coding tool belongs to. Actor can also import *units* which are collections of functions and constants (without side-effects) that can be reused across FUs. Hence, the header of the actor *Algo_ISZigzagOrAlternateHorizontalVertical_8x8* conforms to its textual description. It is included in the texture package of the *sc29/wg11* MPEG-4 part 2 Simple Profile (SP) standard and its signature is composed of two inputs (AC_PRED_DIR, QFS_AC) and one output (PQF_AC), all of them gets integer value of token of size 16.

```
List(type: int(size=SAMPLE_SZ), size=128) buf; // Buffer for Ac coeffs
int(size=8) count := 1;
int(size=9) addr;
function wa() --> int : (count & 63) end
```

Figure 5: State variables and functions of the RVC-CAL implementation for *Algo_ISZigzagOrAlternateHorizontalVertical_8x8*.

The processing of an FU is described according to a sequence of atomic steps called *actions* (Fig. 6), which are defined in the body of an actor. During the execution of an actor, called actor *firing*, one action is selected among the others. The selection of one action may consume input tokens, produce output tokens and change the internal state of the actor. A reference of a port on the input signature of an action indicates that one token is consumed on the given interface when the action is selected. Conversely, a reference of a port on the output signature of an action indicates that one token is produced on the given interface when it is selected. Each port refers to one or several variables that contain the data to be consumed or produced. An action may also be connected to a *label* and contain a body that defines the series of statements to be applied to these variables. Statements include calls to external *functions* declared in the actor. All the actions from the actor in Fig. 6 produce and consume a single token at each firing. However, RVC-CAL places no restriction on token consumption/production. Actions may include the optional semantic *repeat* that allows extending its consumption/production to several tokens.

```
skip: action AC_PRED_DIR:[ i ] ==> //Wait for a valid Inverse Scan
  guard i < 0 end

start: action AC_PRED_DIR:[ i ] ==>
end

read_write: action QFS_AC:[ ac ] ==> PQF_AC:[ buf[addr] ]
guard count < 64
do
  buf[wa()] := ac;
  count := count + 1;
  addr := addr + 1;
end

done: action ==> // Done reading or writing
  guard count = 64
(...)
```

Figure 6: Example of RVC-CAL actions for implementation of *Algo_ISZigzagOrAlternateHorizontalVertical_8x8*.

An action has side-effects on the states of an actor for each firing. These states are represented in RVC-CAL both by state variables (Fig. 5), a Finite State Machine or priorities (Fig. 7). They are used for constraining the sequence order of the actions or for producing state-dependent computations. *Guards*, when present, constrain action firings according to a state variable or a data value from an input port. As such, an action can fire if and if only (iff) its associated guard is *true*. An FSM regulates the action firings according to state transitions. One action can be fired at a time. If multiple actions can be fired in a given time, the action fired is the one with the highest priority if it exists.

By way of example, the actor *Algo_ISZigzagOrAlternateHorizontalVertical_8x8* has several actions; some of them are described on Figure 6. The *skip* and *start* both read one token from input *AC_PRED_DIR*. The action *skip* may fire iff the first data on *AC_PRED_DIR* has a strictly negative value. On the contrary, the action *start* may fire for any value on *AC_PRED_DIR*. The action *read_write* reads one token from *QFS_AC* and produces one token on *PQF_AC*. This action may fire according to *count* and has several side effects on state variables. The *done* action has an empty signature but a conditional firing on *count*.

The FSM on Figure 7 gives the legal firing sequence on these actions. For instance, in its initial state *rest*, a

```
schedule fsm rest :
  rest ( skip ) --> rest;
  rest ( start ) --> read;
  (...)
  full ( start ) --> both;
  both ( read_write ) --> both;
  both ( done ) --> full;;
end

priority
  skip > start;
  done > read_write;
end
```

Figure 7: RVC-CAL FSM and priorities of *Algo_ISZigzagOrAlternateHorizontalVertical_8x8*.

negative value *AC_PRED_DIR* may fire *skip* and *start*. *skip* is the action fired as it owns the highest priory.

3. Implementing RVC specifications

MPEG RVC relies on the two exposed standards to provide an entire specification of their video compression standard with the help of FUs from the VTL and a set of configuration to express the *profiles* of their decoder. The given specifications can be also be derived into proprietary implementations of decoders. Indeed, the strong encapsulation of its FU, along with the network aspect of a configuration, is already an explicit model of execution based on concurrent computing [7]. Each FU is a "black box" component that can be implemented either in software or hardware as long as its external behavior conforms to the reference description in RVC-CAL. As shown in Figure 8, an RVC specification of decoder can be instantiated from a configuration by: (1) executing concurrently all FU implementation represented by the vertices of network (2) following data dependencies rules on their interface represented by the connections of a network.

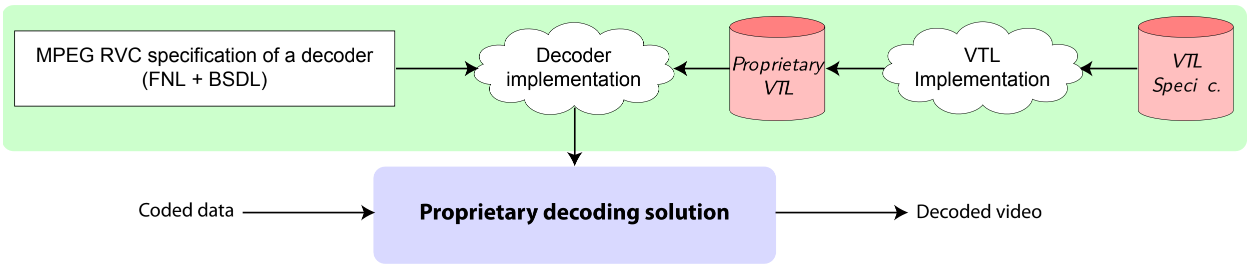

Figure 8: Conceptual process of an RVC decoder implementation.

A case study of implementations from RVC specifications can provide a way for decoder designers to validate the conformance of their proprietary implementations to the original standard. However, RVC specifications cannot be used "as is" to obtain efficient implementations into software platforms. Indeed, given the restricted number of their processing resources, a genuine concurrency of the execution of dozens of processes can rarely be efficiently achieved.

In this section, we draw on usual denotational semantics derived from concurrent process networks to provide automatic, accurate and efficient implementation of MPEG RVC specifications. To achieve this goal, we connect RVC-CAL specification of FUs to a specific denotational that targets a distributed Model of Computation (MoC) called Dataflow Process Network (DPN) [8]. This dataflow denotational does not only provide the key to obtain an accurate translation of a FU specification into concrete implementation; it also allows selecting among many execution models that can execute a configuration according to the number of processing resources a given platform have.

3.1. Dataflow Process Networks

Dataflow programming was primarily inspired by works developed by Kahn in [9]. In this paper, Kahn designs the semantic of a language – denotational semantic – to formalize the behavior of concurrent functional process. This language delimits an environment where a series of deterministic processes communicate asynchronously by passing messages though unidirectional FIFO channels with unbounded capacity. Kahn demonstrates that the overall network of deterministic process, called Kahn Process Networks (KPN), has a deterministic behavior. Thus, a process in a KPN can be described by sequential instructions able to read and write through the communication channels and their execution order has no influence on the output of the system.

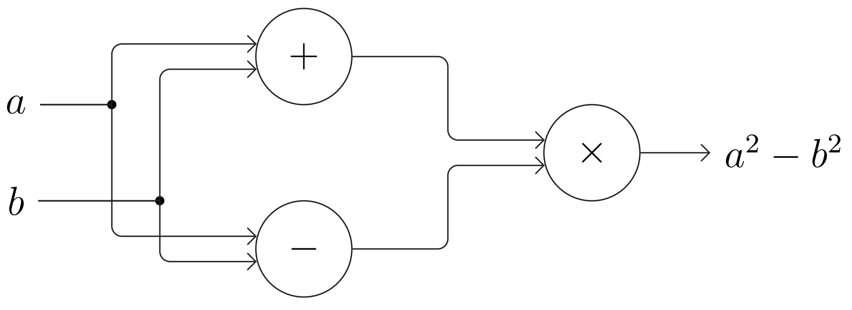

Figure 9: A Kahn Process Network that computes $f : a, b \rightarrow (a + b) \times (a - b)$. .

A KPN is closely related to functional programming where computations are evaluated as a sequence of mathematical functions without side-effects. Computations in a KPN are linked in a block diagram connected by arcs that denote the transmission of packets of information between them. As such, this type of programming is tailor-made to model reactive systems [10], or, in other terms, a system that continually reacts to a flow of data on its input and continually produces flow of data on its output. A graph of a KPN allows breaking down these systems into a collection of communicating operations. These operations are identifiable with the help of a graph which allows a visual representation of the flow of data passing though these operations. Figure 9 illustrates a graph from a KPN that computes the operation $a^2 - b^2$, that is $(a + b) \times (a - b)$. This representation contrasts with imperative programming where applications are described using a series of instructions and these instructions are able to change the overall state of the program.

We define here the *denotational semantic* used by Kahn to describe the behavior of a process on the communication channels as it provides a useful abstraction to formalize concurrent processes. In this notation, each FIFO channel in the network carries at any time a sequence of tokens $X = [x_1, x_2, ...]$, where each x_i is a token. An empty FIFO channel with any token is an empty sequence noted $\perp$. A sequence X that precedes a sequence Y, e.g. $X = [x_1, x_2]$ and $Y = [x_1, x_2, x_3]$, is noted $X \sqsubseteq Y$. The set of all possible sequences is noted S, and S^p is the set of p-tuples of sequences on the p FIFO channels of a process. In other words, $[X_1, X_2, ..., X_p] \in S^p$ represents the sequence consumed/produced by a process. For instance, S^2 corresponds to $s_1 = [[x_1, x_2, x_3], \perp]$ or $s_2 = [[x_1], [x_2]]$. The length of a sequence is given by $|X|$; similarly the length of an element $s \in S^p$ is noted $|s| = [|X_1|, |X_2|, ..., |X_p|]$. Thus, we note $|s_1| = [3, 0]$ and $|s_2| = [1, 1]$.

Based on this denotational semantic, Khan defines a process with m inputs and n outputs as a continuous and monotonic function:

$$F : S^m \rightarrow S^n \tag{1}$$

The monotonicity is a key property of this function as it guarantees that a process can produce a sequence of token without receiving the complete sequence of token on its input. A process in a KPN is triggered when S^m exists on its inputs; it carries on its computation iteratively as long as S^m exists. On the contrary, the process is suspended whilst S^m does not exist on its input.

Dennis [11] and Lee [8] extend the KPN principle by introducing the notion of *firing rules* in processes, which come closer to the notion of *actions* in RVC-CAL. With firing rules, a process can be triggered by several sequences S^m on its inputs. These specific processes are so-called *actors*, the execution of an actor is called *firing* and the composition of actors in a network is a *Dataflow Process Network* (DPN). Lee [8] introduces a new denotational semantic where an actor can have N firing rules:

$$\mathrm{R} = [\mathbf{R}_1, \mathbf{R}_2, ..., \mathbf{R}_N]. \tag{2}$$

A firing rule R_i defines a finite sequence of patterns, one for each m input of the actor:

$$\mathbf{R}_i = [P_{i,1}, P_{i,2}, ..., P_{i,m}] \in S^m. \tag{3}$$

A pattern $P_{i,j}$ defines an acceptable sequence of tokens in R_i on the input j, it is satisfied iff $P_{i,j} \sqsubseteq X_j$ where X_j are the sequence of tokens available on the j^{th} FIFO channels. The pattern $P_{i,j} = \bot$ designates any empty list where any available sequence on input j is acceptable. The pattern $P_{i,j} = [*]$ is acceptable for any sequence *containing at least one token*.

An actor in a DPN fires when at least one of its firing rules is satisfied. In the same manner as a KPN, firing consists of applying a function F as defined in Equation 1. The major difference between these two models is that, whilst KPNs use blocking read when S^m is not present on the inputs of a process; an actor tests sequentially, continuously and in a pre-defined order the validity of firing rules. An actor only consumes data from inputs when a firing rule is satisfied; it does not need to be suspended when it cannot read. Hence, a DPN can be executed by checking sequentially or concurrently the validity of all firing rules contained in its actors. The considerable overhead of context switching caused by the suspension and resumption of processes in most implementations of KPNs is thus avoided.

3.2. From RVC specifications of decoder to Dataflow Process Networks

Dataflow Process networks have been extensively considered in literature to produce efficient execution models based on a concurrent execution. The definition of actors in a network as one firing function and several firing rules allows configuring applications for a wide range of platforms, with no use of synchronization primitives, such as mutex and semaphore. Given this benefits, we develop a Canonical Representation (CR) of RVC-CAL actors based on the DPN semantic, optimized for the execution of RVC specifications. Canonical refers to a representation that follows DPN models in the form of firing rules and firing functions. This CR allows analyzing RVC-CAL actor execution and thus easing their implementation in networks.

A CR of an actor requires identifying all the information that make up the RVC-CAL paradigm. We follow [12] to define an RVC-CAL actor as follow:

- one *identifier* possibly linked to a package,
- *m input ports* that contains S^m,
- *n output ports* that contains S^n,
- one set of $A_i \in A$ *actions* where $A_L \subseteq A$ is the set of actions that contains a *label*.
- one set of *function*, *units* and *procedures*,
- one set of partially and non-reflexive ordered relation (<) that identifies the *priorities* between labeled actions A_L,
- one optional deterministic *Finite State Machine* (FSM),
- one set of constant *parameters* noted Ψ, each of them contains a reference to a ψ value stored in a network,
- one set of *p state variables* noted V.

The set V of state variables contains at any time a sequence $[v_1, ...v_p]$ where v_i is the value of the i^{th} state variables. Some of them are initialized at a value $v_{i,0} \in V_0$, possibly related to a value $\psi \in \Psi$ set by a network.

An action $A_i \in A$ is composed of:

- I_i : an input signature that may refer to all or a subset of the *m input ports* of the actor,
- O_i : an output signature that may refer to all or a subset of the *n input ports* of the actor,
- ϕ_i : a body that describes the computation processed by the action,

- $\mathcal{G}_i$: an optional guard that gives additional conditions to enable an action.

For instance, the FU *Algo_ISZigzagOrAlternateHorizontalVertical_8x8*, depicted on Fig. 1, has two inputs that may contain a sequence S^2. It has one output that may contain S^1. Its RVC-CAL description (Fig. 5) encompasses three state variables, we note $V = [buf, count, addr]$ with $v_{count,0} = 1$. Its set of actions (Fig. 6), which includes $[skip, start, read\_write, done, ...]$, are all contained in A_L. There is also 2 priorities < between skip/start and done/read_write; and one FSM. The action *skip* is noted with $I_{skip} = [AC\_PRED\_DIR]$, $O_1 = [\emptyset]$, it has a body ϕ_{skip} with several side effects on V and a guard condition $\mathcal{G}_skip$.

An RVC-CAL actor, by its construction in the form of actions, can be considered as a particular case of a DPN. Each of the N actions of an actor is a functional process such as:

$$\text{F} = [f_1, f_2, ..., f_N] \tag{4}$$

with localized side effect on the actor state. We define the notation Σ for the set of states in the actor and we represent an action $A_i \in A$ with one *firing function*:

$$f_i : \Sigma \times S^m \rightarrow \Sigma \times S^n \tag{5}$$

that relates a state and m sequences of tokens from the inputs of the actor to a new state $\sigma \in \Sigma$ and n sequences of output tokens. This notation respects the DPN assumption as, from an interface point of view of the actor, F is a Kahn Process, with no side effect on other processes as defined in Equation 1.

One firing rules f_i is triggered by a corresponding *firing rule*:

$$\mathbf{R}_i = [P_{i,1}, P_{i,2}, ..., P_{i,m}] \in S^m,\ \Sigma_i \in \Sigma \tag{6}$$

that defines the acceptable patterns for m sequences of input tokens and the set Σ_i of acceptable states in the actor.

The overall states Σ in the actor can be modeled in RVC-CAL either by values from the state variables V or by the current state of the FSM. We use the following conventional notation to describe an RVC-CAL FSM:

- A_L: an input alphabet corresponding to the label of action;
- S: a finite non-empty set of state S with $s_0 \in S$ an initial state;
- $\delta : S \times A_L \rightarrow S$: a state transition function that relates a state and a label to a new state in the FSM.

The body ϕ_i of an action A_i carries out the computation from S^m to S^n and generates side effects on V. Fitting together δ and ϕ functions allow retrieving the complete firing function f_i for an action A_i:

$$f_i : \begin{cases} \phi_i : V \times S^m \rightarrow V \times S^n,\ and \\ \delta : S \times A_L \rightarrow S. \end{cases} \tag{7}$$

Actions may also call external procedures and functions in the actor which are considered as a part of this ϕ_i function. The *input signature* I_i of an action A_i gives the size of the sequence $|S^n| = [|s_1|, |s_2|, ..., |s_n|]$ on which ϕ_i has to process the computation. The size of the resulting sequence is given by the *output signature* O_i. In other terms, for any input sequence $s \in S^m$ and any values in V, $|S^n| = |\phi_i(s, V)|$.

The *input signature* I_i also gives a subset of one firing rule in the actor. It sets the minimum size of S^m to enable the corresponding firing function; we note this information $|I_i|$ that gives $[|P_{i,1}|, ..., |P_{i,m}|]$. A port j with a *repeat* value at n referenced in the input signature I_i has $|P_{i,j}| = n$. Similarly, a port j in I_i with no *repeat* has $|P_{i,j}| = 1$. A ports of an actor not referenced in I_i correspond to $P_{i,j} = \perp$.

The *guard* gives the following predicate:

$$\mathcal{G}_i = [P_{i,1}, ...P_{i,m}] \in S^m, V_i \in V \tag{8}$$

where $V_i = [v_{(i,0)}, ..., v_{(i,n)}]$ defines a set of acceptable values in the actor state variable and $v_{(i,j)}$ is the set of acceptable values for j^{th} state variable. A state variable not referenced in one guard states that all of its values are acceptable. Similarly, a port j referenced in I_i but not in the guard $\mathcal{G}_i$ is a series of wildtokens $[*, *, ..., *]$ of length $|P_{i,j}|$; in other terms, $P_{i,j} \sqsubseteq X_j$ is valid for a sequence X_j with length greater than $|P_{i,j}|$.

Retrieving the full firing rule $\mathbf{R}_i$ for a given action A_i corresponds to fit together the information from I_i, G_i and δ from the FSM. Thus, a firing rule $\mathbf{R}_i$ is valid for an action A_i when:

$$\mathbf{R}_i = \begin{cases} \forall X_j \in S^m : \ |X_j| \geq |P_{i,j}|, \ P_{i,j} \sqsubseteq X_j, \ and \\ \forall v_j \in V : v_j \in V_i, \ and \\ \exists \ (t \times A_i) \in \delta. \end{cases} \tag{9}$$

X_j is the sequences from the j^{th} ports in S^m, v_j is the value of the j^{th} state variable in V, and $t \in S$ is the current state of the FSM. All these conditions must hold true to valid a firing rule. Moreover, condition $|X_j| \geq |P_{i,j}|$ can be equivalent to condition $P_{i,j} \sqsubseteq X_j$ when $P_{i,j} = [*, .., *]$. In this case, $|X_j| \geq |P_{i,j}|$ is considered as a necessary and sufficient condition to validate the pattern: it implies $P_{i,j} \sqsubseteq X_j$. For example, considering that X_a denotes the sequence contained on the port *AC_PRED_DIR* and t is the current state of the FSM, the test of the firing rule from the *start* action (Fig. 6) can be written as follows:

$$\mathbf{R}_{start} = [\exists \ X_a, \ t \mid \ |X_a| \geq 1, \ (t \times A_{start}) \in \delta]. \tag{10}$$

Actor may also have non-deterministic behavior as they are allowed (1) to test inputs for emptiness, (2) to be internally non-deterministic [8]. A case of non-deterministic behavior happens when an actor has several firing rules valid for a given sequence $s \in S^m$ and for a state $\sigma \in \Sigma$. The action fired is the one which has enough room available on its outputs to store S^n. As a way of example, the *read_write* and *done* action on Figure 6 can have non-deterministic behavior in the case where 1 token is available on *QFS_AC* and the FSM is in the state *both*.

Priorities in RVC-CAL resolve this non-deterministic behavior. They allow setting a partial order to test the N firing rules of an actor which are organized in a 2-dimension array as follow:

$$\mathrm{R} = \begin{pmatrix} \mathbf{R}_1 > \mathbf{R}_2 > \mathbf{R}_3 \\ \mathbf{R}_4 > \mathbf{R}_5 \\ ... \\ \mathbf{R}_N \end{pmatrix}. \tag{11}$$

Firing rules are ordered by descending priority on each line of the array. As such, a valid firing rule invalidates the firing rules for actions with a lower priority. Two actions with no notion of priority – i.e. on the same column – are have a testing order that depends on the implementation of this CR.

3.3. From Dataflow Process Networks to execution models

As referred in [8], DPNs have natural execution models due to the breakdown of a process into a sequence of actor firings. Following the exposed RVC-CAL denotational, a DPN of RVC-CAL actor is composed of a set of firing rules $\mathbf{R}$ where each firing rule may encompass a partially ordered subset of firing rules $\mathbf{R}_i$ according to their respective priorities. Instead of the context switching found in typical concurrent execution models, the overall application can be executed by continuously scheduling actor firings according to firing rules.

As RVC-CAL actors may have data-dependent behaviors, the scheduling can be only done at run time in the general case. It implies that firing rules should be able to evaluate the absence or presence of tokens from/to the FIFO channels at any time during the execution. It also requires "*peeking*" at values from a FIFO to check if a sequence X_j from a FIFO j matches a given pattern $P_{i,j}$ for a firing rule $\mathbf{R}_i$. The interactions between firing rules and FIFO channels can be summarized with the help of two functions which:

1. Gets the number of tokens available in a FIFO j. This function allows evaluating for one firing rule $\mathbf{R}_i$ if the contained sequence X corresponds to the condition $|P_{i,j}| \leq |X|$.
2. "Peeks" at a fixed number of tokens from a FIFO j without consuming. It allows evaluating for a firing rule $\mathbf{R}_i$ if the contained sequence X corresponds to the condition $P_{i,j} \sqsubseteq X$.

The monotonic property of firing functions (Eq. 1) allows the application to work on a FIFO with bounded sizes [9]. Firing a function requires to consume a fixed number of token from FIFO channels to construct S^m and to store S^n into FIFO channels. Working on bounded FIFO requires also requires an additional condition on firing functions that verify, before a firing, if there are enough rooms in FIFO channels to store the produced sequence S^n. The interactions between firing rules and FIFO channels can be summarized with the help of three functions which:

1. Gets amount of rooms available in a FIFO j. It allows evaluating if a sequence of token $X_j \in S^m$ can be stored in an FIFO j of size l that already contains a sequence X; in other terms it evaluates if $|X_j| \leq |X| - |l|$.
2. Consumes tokens from a FIFO j. Consuming means reading and clearing a sequence of tokens X_j from the FIFO memory.
3. Stores tokens into a FIFO j. It involves a copy of the sequence of token $X_j \in S^n$ in the FIFO memory.

The instantiation of a DPN involves the creation of instances of RVC-CAL actors on each vertex of the network. Each instance encompasses its own state Σ which is initialized at state $\sigma_0 = [V_0, s_0]$ and where values ψ of parameters Ψ may also be propagated from the network to V_0. An execution model, also called *scheduling strategy*, is finally associated to the DPN for its execution on a platform. It exists a wide variety of execution models for DPNs, this is due to the fact that DPNs do not over specify an algorithm the way non-declarative semantics do.

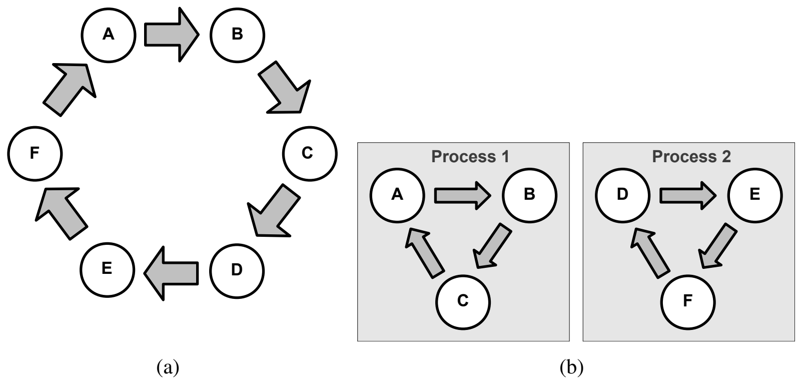

Figure 10: Scheduling of 6 actors in a DPN with a round-robin strategy by (a) one process (b) two processes.

The simplest execution model consists of testing the validity of the firing rules for each actor with a round-robin strategy. This strategy can be carried out, as illustrated in Fig. 10, by one process or several concurrent processes by using protected FIFO. It has low complexity and it is simple to implement; but it may involve a low chance of success between the test and the validation of firing rules as it makes no assumption of the network topology. More advanced technologies, such as data-driven strategy, demand-driven strategy, or mixed data-driven/demand-driven, all referenced in [13], have already been experienced on a DPN with RVC-CAL actors [14].

For many signal processing applications, the firing sequence can also be determined statically, i.e. at compile-time. The subsets of classes in DPN for which this is always possible are called Synchronous DataFlow (SDF) and Cyclo-Static Dataflow (CSDF). In a SDF network, actors always have a single firing rule, which is valid for any sequence S^m on their inputs [15, 16]. In other words, an SDF actor is enabled by a fixed number of tokens at each input and has no data-dependent behavior. In the case where an actor has several firing rules, this actor is SDF if all its firing rules have the same consumption [17], i.e. for $\forall \mathbf{R}_A \in R$ and $\forall \mathbf{R}_B \in R$:

$$|\mathbf{R}_A| = |\mathbf{R}_B|. \tag{12}$$

The firing function of an SDF actor must also produce a fixed number of tokens when it fires:

$$|F(s_a)| = |F(s_b)|. \tag{13}$$

for any $s_a \in S^m$ and $s_b \in S^m$. CSDF [18] extends this property by allowing the number of tokens produced and consumed by an actor to vary cyclically. This variation is then modeled with a state in Σ that rules firing and returns to its initial value after one period.

RVC-CAL actor may semantically express SDF and CSDF behaviors. For instance, an actor composed of one action is in essence an SDF actor. In this way, the transformation presented in the previous section helps exposing static behaviors of RVC-CAL actors composed of multiple actions by validating for each i, j of the N actions (1) Eq. 12 for each $|P_{i,k}| = |P_{j,k}|$ in Eq. 6 with $0 < k < m$ and m being the number of input ports of the actor (2) Eq. 12 for each $|\phi_i(s)| = |\phi_j(s)|$ given by the output signature $|O_i|$ and $|O_j|$ respectively in Eq. 7. More complex descriptions of RVC-CAL actors composed of multiple actions with different sizes on their signatures can also expose a static behavior at the execution. However, this static behavior may be difficult to guess semantically as they depend on state conditions (from guard or FSM) or priorities. Some classification algorithms, such as [19], allow detecting static behaviors of DPN actor using run-time interpretation. They analyze the reaction of an actor to different stimuli of sequence of tokens. If an actor continuously produces a fixed/cyclic amount of token for any input sequence of increasing size, then the actor is either SDF or CSDF.

The benefit of a network only composed by SDF actors is that the execution sequence of all actors can be determined with a set of balance equations relating the production and consumption from each actor. This property allows avoiding the overhead of run-time scheduling and optimizing the size allocated for each FIFO. Moreover, some algorithms can map and schedule SDF graphs onto multi-processors in linear time with respect to the number of vertices and processors [20].

Many other execution models extend this behavior by allowing a trade-off between data-dependent behavior and static scheduling analysis. The Parameterized DataFlow [21], which includes Boolean DataFlow [22] and Quasi-Static DataFlow (QSDF) [23] models, is a subset of the DPN model that allow statically analysis of the firing of actors in a network, so that only data-dependent operations are scheduled at runtime. For a more in-depth into scheduling strategies, [8] gives a comprehensive comparison of execution models with their different strengths and weaknesses. Works have also already been initiated in [24] to adapt the scheduling process of a DPN according to the behavior of each of its actor.

4. Reconfigurable implementation: the RVC decoder case study

The dataflow mechanisms defined in the previous section make it possible to reach efficient implementations of RVC specifications whatever the platform targeted. Such implementations remain conformant to RVC standards as it conserves a strict separation between coding tools representations and the execution model to be applied to a configuration of a decoder. Moreover, each coding tool implementation keeps the same I/O compared to their specifications in MPEG-C part 4.

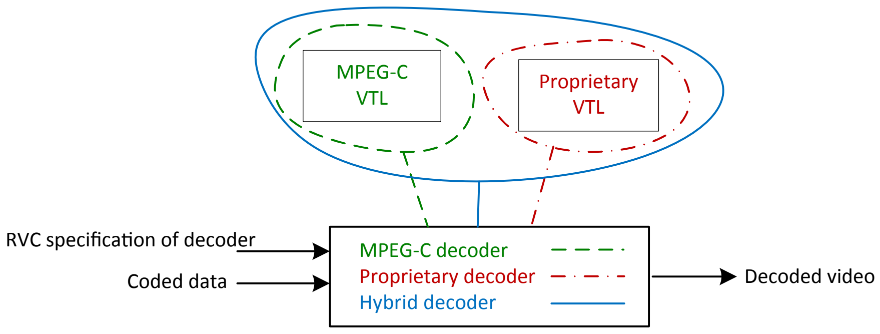

Figure 11: Conceptual view of an RVC dynamic decoder in MPEG-4 part 4 standard.

Obtaining actual reconfigurable implementation of decoder specifications can only be achieved by introducing dynamic behavior inside the implementation mechanisms. One objective of the MPEG-B part 4 standard is to provide for end-users a RVC decoder (Fig. 11) that can decode any coded bitstream as long as an RVC specification of the requested decoder is coupled with it. As such, an RVC decoder aims to remove any interoperability problems that can appear between a content provider and several receivers. It can also support proprietary VTLs which may be used to produce proprietary or hybrid specifications of decoder that include coding tools not standardized in MPEG-C part 4. Indeed, as network representations are modular, they support any kind of coding tools and they facilitate the reconfiguration of a decoder by only modifying the topology of the network. This RVC decoding approach opens the way to new decoder functionalities which can change its decoding computation during the decoding process, as presented in [25].

The support of dynamic implementation in RVC decoder requires to provide automatic implementations of RVC specifications and to be able to change this implementation according to a new network representation. We present in this section an implementation example based on virtual machine mechanisms.

4.1. Automatic synthesis of RVC decoders

Synthesis tools that provide automatic implementation of RVC specification already exist in the RVC framework. The RVC-CAL reference language, used for MPEG RVC specification, has been designed as a subset of the CAL Actor Language (CAL). Compared to CAL, RVC-CAL retains a high level of abstraction to describe actors, but reduces its expressivity on types, operators and functionality that cannot be easily integrated on hardware platforms. However, synthesis tools that support the CAL languages can be used to manage RVC specifications of decoders. As a way of example, CAL is historically supported by a Java interpreter integrated in Ptolemy II [26], in Moses [27] and in OpenDF [28].

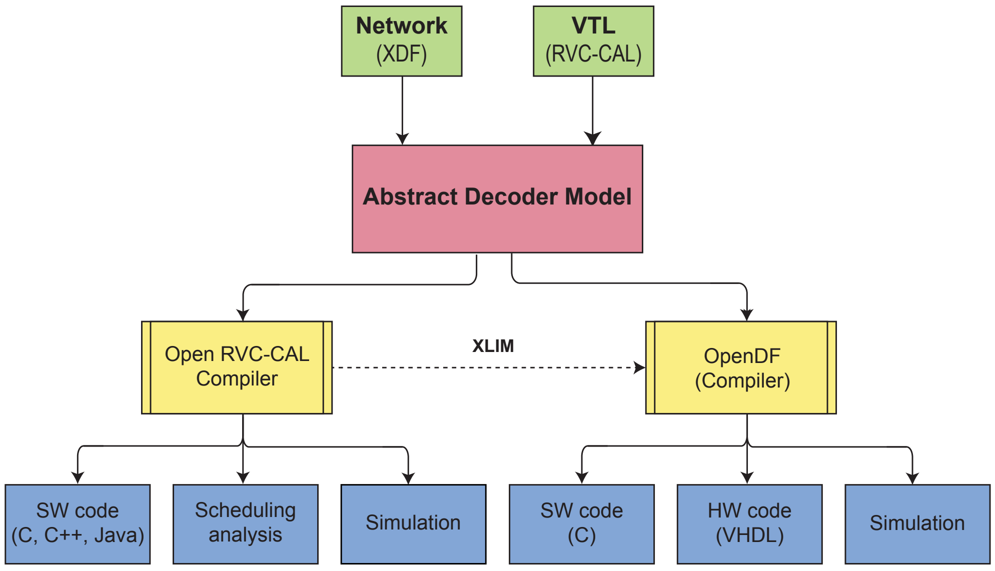

Figure 12: “Static” synthesis for MPEG RVC specification.

OpenDF has a specific feature that uses, on the top of its interpreter, an Intermediate Representation (IR) in XML Language-Independent Model (XLIM) [29]. This IR is close to low-level imperative languages and allows generating HDL representation of CAL network through an XLIM to HDL converter called OpenForge [30]. OpenDF can additionally be coupled with an XLIM to C converter to target software platforms and particularly ARM-based platforms [31]. As illustrated in Figure 12, these tools are used in the MPEG RVC framework to provide C or HDL representations of MPEG RVC specifications.

The Open RVC-CAL Compiler (Orcc), compared to OpenDF, provides a complete Integrated Development Environment (IDE) dedicated for designing, analyzing and transforming RVC specifications. It brings together both for networks and for RVC-CAL actors an editor, a compiler, a debugger and a classifier that retrieves the SDF, CSDF and QSDF behavior of RVC-CAL actors. It also encompasses a specific XLIM representation compiler to bridge the gap with the OpenDF framework.

These synthesis tools are used in MPEG RVC to generate C or VHDL implementations from RVC specifications of decoders. The resulting decoder may then be compiled by a developer to produce implementations in native code; this native code is finally transmitted ”as is” to the targeted platforms. As such, the resulting decoder does not require additional tools to be executed at the customer’s side, but it does not provide the dynamicity on implementations required to produce a RVC decoder.

A dynamic implementation can only be provided by a compiler that works at the customer side. We use in our case study *virtual machines* as they incorporate efficient compilation mechanisms, such as Just-In-Time (JIT) compilation, that can significantly reduce delays to obtain implementations of decoder. Virtual machines dedicated to the execution of signal processing systems are described in the literature [32, 33, 34]. Unfortunately, the majority

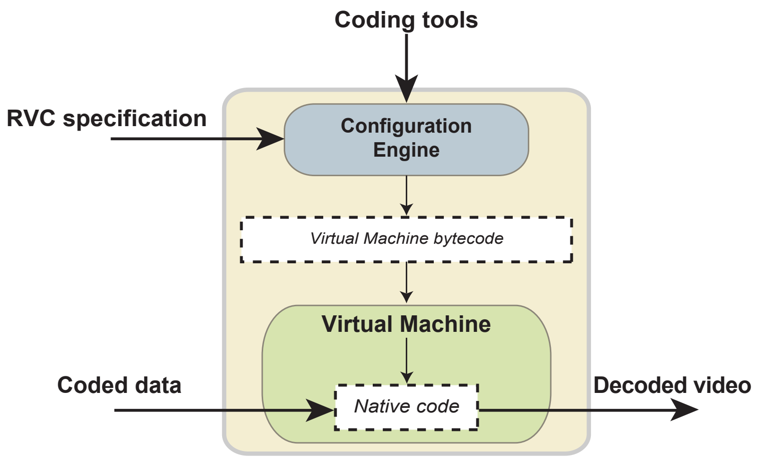

Figure 13: Reconfigurable implementation of MPEG RVC decoder based on virtual machines.

only loosely follows dataflow mechanisms introduced in DPNs. For our case study, we propose an actor-oriented bytecode that strictly draws on the denotational introduced in section 3.2 and the principles typically integrated into virtual machines based on imperative languages. As such, we propose the structure in Figure 13 that extends bytecode of imperative languages to support RVC specifications.

Following this structure, a reconfigurable implementation of RVC specifications becomes a three step process:

1. A *configuration engine*, built on the top of a virtual machine, acquires a network from an RVC specification and selects the required coding tools from among a proprietary or standard VTL.
2. It produces an implementation of the corresponding decoder and a model of execution in an imperative bytecode form according to the virtual machine environment.
3. The virtual machine gets and translates the produced bytecode into native code for its execution on the platform. This execution can finally process data coded to produce the decoded video.

A configuration engine may take as input RVC-CAL specifications of coding tools. However, translating their textual form into bytecode can be a pointless and wasteful process. Based on the work presented in the previous section, we develop a specific bytecode optimized for the configuration engine and virtual machines in a dataflow context.

4.2. *Extending imperative bytecodes to actor-oriented bytecodes*

Bytecodes of virtual machine are derived in the general case from imperative languages. They form of an *instruction set* that composes a *Virtual-Instruction Set Architecture* (V-ISA). This V-ISA is designed to produce an efficient execution on virtual machines, as such they encompass binary instruction codes, constants and references that result from the semantic analysis of types, scopes and nesting depths of program objects. They also usually provide a better compression rate and better performance than direct interpretation or compilation of source code.

```
define i32 @f (i32 %x) {
  %0 = load i32* @A     // Load value a from global variable A in register 0
  %1 = load i32* @B     // Load value b from global variable B in register 1
  %2 = mul i32 %x, %0   // Multiply x by a and store the result on register 2
  %3 = add i32 %1, %2   // Add b to register 2 and store the result on register 3
  ret i32 %3            // Return end result fron register 3
}
```

Figure 14: Register instructions from Low-Level Virtual Machine (LLVM) instruction set that computes $f : [a, b] \times x \rightarrow [a, b] \times [a \times x + b]$.

	Metadata	V-ISA
MCR for a DPN	S^m, S^n, R	S^m, S^n, F

Table 2: MCR bytecode of an actor.

The two most widely used V-ISA from procedural languages mimics programs with a series of (1) *stack-based instructions* that add and remove data to/from a stack or (2) *registers* instructions, that load and store values from/to virtual registers. For example, Figure 14 gives a textual representation of a procedure that realizes the operation $f : [a, b] \times x \rightarrow [a, b] \times [a \times x + b]$ using only register instructions from the *Low-Level Virtual Machine* [35]. In this code, $[a, b]$ are values respectively stored in global variables A and B, x is a parameter of function f, $\%0, \%1, \%2, \%3$ are virtual registers that stores the intermediate values for the computation of the f function. Most bytecodes also support metadata – meta information on V-ISA – to extend their support to high-level programming languages. In the context of object-oriented programming, metadata may be used to give extra information on data types, referring them to classes, attributes or relationships with other classes. Their corresponding virtual machine, illustrated on Fig. 15, uses a loader to translate metadata into data structures. The compiler translates the V-ISA into native code and communicates with this loader to link data structures with the native code.

For our case study, we use metadata to extend bytecodes toward the support of actors. We refer to this new bytecode as a Minimal and Canonical Representation (MCR) of actor. The term *Canonical* introduces the DPN form of RVC-CAL actors introduced in section 3.2. The term *Minimal* adds a constraints for minimizing the transformations to be applied on the representation to obtain a reconfigurable implementation with virtual machines. As such, this representation can be viewed as an enhanced bytecode for virtual machine that provides explicit concurrency between algorithms. It does not only virtualize the ISA of platforms, as bytecode does normally, but also abstracts the computation resources involved in platforms. The configuration engine from Figure 13 plays the role of a virtual machine loader that transforms the MCR of several actors coupled with a network into an entire bytecode representation form for the underlying virtual machine.

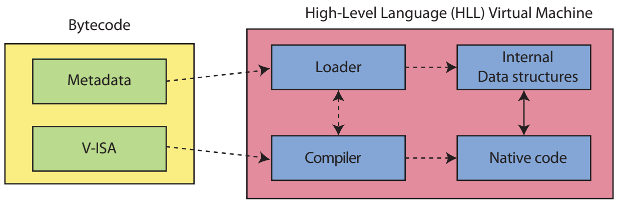

Figure 15: Virtual Machine for High-Level Languages.

The key property of the MCR is to isolate the computations achieved by actors from their implementation environment. In section 3.2, we define an actor as a firing function (Eq. 1) that requires *communication mechanisms* to retrieve S^m and produce S^n from/to the network. The firing function conserves its computation whatever the network that surrounds the actor; it has thus a direct equivalence with functions and parameters of a V-ISA. The *firing rules* from Equation 2 allow retrieving the execution sequence of the actor firing from a network. They also add *data-dependent* and *non-deterministic* behaviors on this execution sequence. The transformational aspect of an actor is thus based on the manner to test firing rules and to produce/consume sequence of tokens on the communication channels. Starting from this observation, we propose in Tab. 2 a separation between metadata and bytecode instruction for an actor with m inputs and n outputs that conserves the encapsulation of an actor. S^m, S^n are both depicted with metadata and V-ISA instructions as they help for the practical application of F and R and they allow propagating a sequence of tokens from the network to the input of the actor and from the output of an actor to the network.

The MCR is extended to RVC-CAL actors by adding the notion of side-effects (Eq. 5) on the actor state from the firing functions. Actor state, previously noted Σ, can be depicted with a set of global variables in the V-ISA that represents the set of state variables V in the actor and the current state t of the FSM. We deduce the MCR of an RVC-CAL actor in Tab. 3. References from metadata to V-ISA are depicted with a single line in Table. The actor's name is only informative, it allows identifying an actor for its implementation in a network. The set of global variables in

	Metadata	V-ISA
MCR for an Actor	actor's name the m input ports that create S^m the n output ports to store S^n all $R_i \in R$ ordered by priorities	 S^m S^n $[V, t]$, S^m
MCR for an action $A_i \in A$	R_i, $\|I_i\|$ f_i, $\|O_i\|$	$\mathcal{G}_i(V, S^m)$ $\phi_i(V, S^m)$, $\delta(t \times A_i)$, S^n

Table 3: MCR bytecode of an RVC-CAL actor.

the V-ISA that stores the actor state and the input sequence S^m are referenced by firing rules to allow testing their values. The functional computation f_i of an action A_i references two procedures ϕ_i and δ_i in the V-ISA that follow the function side-effect on the actor state and the produced sequence S^n. The minimization aspect of the MCR is obtained by reducing the number of transformations to be applied on S^m, S^n, F and R in the actor. All identified functions in the MCR, such as $\mathcal{G}_i$ identified from *guards* or $\delta(t \times A_L)$ from the *FSM*, are thus directly provided in V-ISA form.

This set of metadata aims to be converted by the configuration engine into V-ISA according to the characteristics of the running platform, the implementation network and a given scheduling strategy. We give in Fig. 16 an implementation example of the MCR that reproduces the interaction of the actions *skip* and *start* on the network and their data-dependent enabling according to tokens on port *AC_PRED_DIR*. As such, we provide several communication mechanisms in the configuration engine that allow interacting with FIFO channels:

1. *Token*: a function that gets the depth of a FIFO, i.e. the number of tokens it contains.
2. *Peek*: a function that peeks at one value of token from a FIFO and returns it without consumption.
3. *Read*: function that consumes one tokens from a FIFO and returns its value.

The configuration engine incorporates the firing rules from *skip* and *start* into a series of conditional statements. The *else if* statement depicts the priority between these two actions. The function $\mathcal{G}_{skip}$ evaluates the condition $i < 0$ from the guard of *skip*. Firing the functions from action *skip* or *start* calls the execution of their respective body ϕ_{skip} and ϕ_{start} on the value consumed from *AC_PRED_DIR*. It also changes the current state t of the FSM with the function δ, which takes the FSM state and one action label and returns a new state.

```
**if** $t = rest$ and $Token(AC\_PRED\_DIR) > 1$ **then**
    $i \leftarrow Peek(AC\_PRED\_DIR)$
    **if** $\mathcal{G}_{skip}(i)$ **then**
        $s \leftarrow Read(AC\_PRED\_DIR)$
        $\phi_{skip}(s)$
        $t \leftarrow \delta(t, skip)$
    **end if**
**else if** $t = rest$ and $Token(AC\_PRED\_DIR) > 1$ **then**
    $s \leftarrow Read(AC\_PRED\_DIR)$
    $\phi_{start}(s)$
    $t \leftarrow \delta(t, start)$
**end if**
```

Figure 16: Pseudo-code that fires actions *skip* and *start* from actor *Algo_IS_ZigzagOrAlternateHorizontalVertical_8x8*.

4.3. Actor-oriented bytecode for the LLVM

We provide in this section a concrete example of MCR based on the LLVM bytecode. We develop a solution structured in 2 parts as a part of the Orcc open-source project[1] and LLVM open-source project[2]:

[1]Orcc project: http://orcc.sourceforge.net

[2]LLVM project: http://llvm.org

1. *A MCR compiler*: based on the Orcc compilation framework, it starts from RVC-CAL specifications of FUs to produce the corresponding MCR in LLVM form.
2. *A virtual machine for MCR*: based on the LLVM compilation framework [36], it starts from the MCR of FU and a network representation of an application and provides a dynamic implementation that fits the running platform.

For our case study, we integrate the virtual machine into an *RVC module* for the GPAC multimedia framework [37] to form an *RVC decoder* (Fig. 17). It uses the GPAC player to analyze a media container complaint with RVC and extracts the network representation of the decoder required to decode a content. The network, conformant with MPEG-B part 4, is then sent to the RVC module. The RVC module constructs the corresponding implementation, processes the content and sends the decoded video to display.

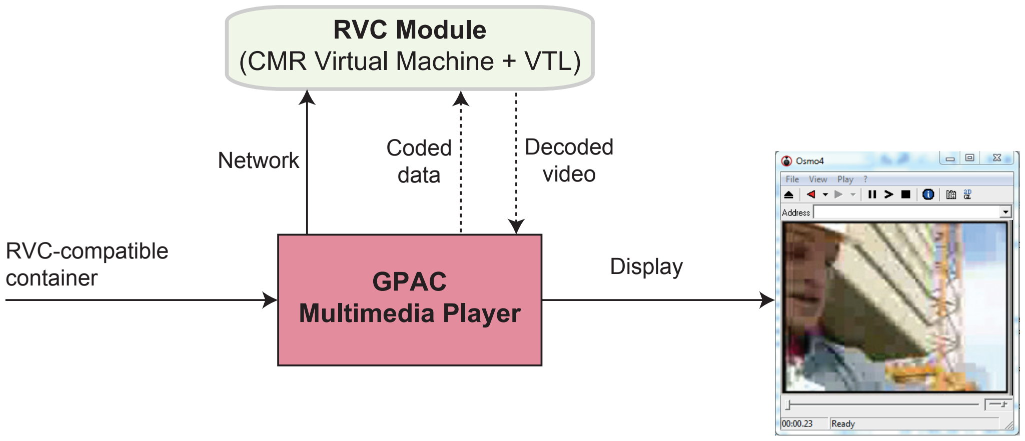

Figure 17: RVC decoder based on the GPAC multimedia framework.

More information about the implementation of the MCR compiler and its virtual machine can be found in [38] and [39]. We tested this framework on two dataflow representations of decoders standardized in MPEG RVC. The first representation corresponds to the *Simple Profile* (SP) specification of the MPEG-4 part 2 standard and the second representation corresponds to *Constrained Baseline Profile* (CBP) specification of the MPEG-4 part 10 Advance Video Coding (AVC) standard. Details of these two dataflow representations are given in [40].

Table 4 gives the size of the VTL at its current state of standardization that includes coding tools from MPEG-4 part 2 *SP* and MPEG-4 part 10 *CBP*. The MCR in LLVM of the VTL (817Kb) is approximately 1.5 lighter than its RVC-CAL specification (1.4 Mb). Their corresponding networks that should be sent with a coded content are respectively 25 Kbytes and 101 Kbytes, they are compressed respectively to 3 Kbytes and 6 Kbytes into media containers using gzip. We use as a comparison the version 2.3 of the MPEG-4 part 2 SP reference software from Microsoft written in C. We compile it with Visual Studio 2010 with full optimizations activated. The resulting application is 614 Kbytes, as such the compressed network representation of the same decoder is less than 1% compared to the size of a the reference software. We also use the JM reference software of MPEG-4 part 10 CBP, which is delivered by the Fraunhofer. A compilation in the same environment gives an application of size 480 Kbytes, the compressed networks of the same application is also 1% of the size of the reference software. As such, embedding network representation instead of full representation of decoder inside media container allow a reduced impact on its size.

	MCR (LLVM)	Raw network (FNL)	Compressed network (gzip)
MPEG-4 part 2 *SP*	198 Kb	25 Kb	3 Kb
MPEG-4 part 10 *CBP*	616 Kb	101 Kb	6 Kb

Table 4: Size of the MPEG-4 part 2 *SP* and MPEG-4 part 10 *CBP* from the VTL in MCR using the LLVM bytecode with their corresponding network in raw and compressed format.

The second experiment in Tab. 5 shows the performance of the implementation on a Core2Duo (X64) processor at 2.40 GHz running on Windows 7 using a round-robin strategy. The test sequences used are the validation sequences provided by the MPEG consortium. For MPEG-4 part 2 SP, it is the sequence *foreman* of size CIF (352 × 288) composed of 300 frames at 30 frame/sec. For MPEG-4 AVC CBP, it is sequence *combine* of size QCIF (176 × 144) composed of 1700 frames at 30 frame/sec. The testing results are taken from an Intel E6600 Core2 Duo processor at 2.40 GHz running on Windows XP.

	1 thread	2 threads	Gain
MPEG-4 part 2 SP	144 fps	265 fps	1.7
MPEG-4 part 10 CBP	50 fps	95 fps	1.9

Table 5: Performance for CIF (352 × 288) video for MPEG-4 part 2 Simple Profile and MPEG-4 part 10 Constrained Baseline Profile using *round-robin strategy* with *1 thread* and *2 threads*.

One and two POSIX threads were respectively used to contain a fixed distribution of the actors from the two networks. The execution on two threads increases the frame decoding rate by 1.7 and 1.9 times on the MPEG-4 part 2 SP and PEG-4 part 10 CBP respectively. These preliminary results prove the advantage of using DPN representation in a concurrent execution context. However, it also shows the inefficiency of a round-robin strategy for single core execution. Indeed, the overhead produce by the scheduler induces reduced by 10 the frame rate for a single core execution, compared to the two reference software from Microsoft and Fraunhofer previously exposed. Nevertheless, the gain incurred by using 2 threads compensates the drawback of this overhead and the MCR can include any execution model, as one presented in [14]. These works show that a distribution of the execution on 4 processes of the same applications using a round-robin strategy can obtain gains up to 3.36 compared to a single core execution. Moreover, by using advanced strategies such as data-driven/demand-driven strategies, the performance of these applications can be increased by 5 compared to the round-robin strategy on a single core unit.

5. Conclusion

This paper demonstrates that it is possible to produce accurate and reconfigurable implementations of media decoders on practical platforms, starting from RVC specification. This implementation can be made by following the denotational semantic introduced in the Kahn Process Network and the more efficient denotational semantic used to describe Dataflow Process Network (DPN). The resulting MCR based on DPN does not over-specify the implementation of decoder, which means that any execution model given in the literature can be associated with the decoder representation. However, a practical implementation using the LLVM framework shows that simple approaches for execution model, such as a round-robin strategy, are sub-optimal compared with imperative languages for an execution on a single core execution.

Nevertheless, when a concurrent execution is required on the implementation, the DPN root of the MCR allows avoiding the massive context switching that can occur by using other models of execution. It provides an execution natively scalable to the number of processing units of a given platform and it can support automatic transformation to more efficient, but less expressive, models such as Synchronous DataFlow (SDF) or Quasi-Static DataFlow (QSDF). The concrete implementation of MCR in LLVM also proves the compact representation of the MCR.

The multi-core ability of the MCR is intended for current and next architectures that will be incorporated in multimedia terminals. The DPN are widely used across many application domains including Digital Signal Processing (DSP), Graphics Processing Unit (GPU), many-core and massively multi-core systems. In this sense, works are currently in process to port the MCR of RVC-CAL actors to GPU, DSP and Multi-Processor System on Chip (MPSoC). The overall approach presented in this paper is also not limited to video coding applications. The separation between network and processing tools provides a useful abstraction to design any signal processing applications. At the present time, the RVC specification has already been extended to audio (Reconfigurable Audio Coding) and 3D (Reconfigurable Graphic Coding), which together form the Reconfigurable Media Coding (RMC) framework.

6. Acknowledgments

We would like to thank the entire Orcc team, but especially Matthieu Wipliez, Nicolas Siret, Maxime Pelcat and Herve Yviquel, for their work on this compiling infrastructure that asserts the base for our experiments. We would also like to gratefully acknowledge helpful comments and suggestions from Iain Richardson and Maja Bystrom at OneCodec Ltd.

References

[1] M. Mattavelli, I. Amer, M. Raulet, The Reconfigurable Video Coding Standard [Standards in a Nutshell], Signal Processing Magazine, IEEE 27 (3) (2010) 159 –167.
[2] ISO/IEC 23001-4, MPEG systems technologies – Part 4: Codec Configuration Representation (2011).
[3] ISO/IEC CD 23002-4, Information technology - MPEG video technologies - Part 4: Video tool library (2010).
[4] ISO/IEC 23001-5, Information technology - MPEG systems technologies - Part 5: Bitstream Syntax Description Language (2008).
[5] M. Raulet, J. Piat, C. Lucarz, 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, IEEE, 2008, pp. 293–298.
[6] J. Eker, J. Janneck, CAL Language Report, Tech. Rep. ERL Technical Memo UCB/ERL M03/48, University of California at Berkeley (Dec. 2003).
[7] M. Ben-Ari, Principles of concurrent and distributed programming, Addison-Wesley Longman, 2006.
[8] E. A. Lee, T. M. Parks, Dataflow Process Networks, Proceedings of the IEEE 83 (5) (1995) 773–801.
[9] G. Kahn, The semantics of a simple language for parallel programming, in: J. L. Rosenfeld (Ed.), Information processing, North Holland, Amsterdam, Stockholm, Sweden, 1974, pp. 471–475.
[10] L. Aceto, Reactive systems: modelling, specification and verification, Cambridge Univ Pr, 2007.
[11] J. B. Dennis, First version of a data flow procedure language, in: Proceedings of the Colloque sur la Programmation, Vol. 19 of Lecture Notes in Computer Science, Springer, 1974, pp. 362–376.
[12] J. Eker, J. Janneck, A Structured Description Of Dataflow Actors And Its Application (May 2003).
[13] T. M. Parks, Bounded Scheduling of Process Networks, Ph.D. thesis, Berkeley, Berkeley, CA, USA (1995).
[14] E. Yviquel, E. Casseau, M. Wipliez, M. Raulet, Efficient Multicore Scheduling Of Dataflow Process Networks, in: Signal Processing Systems (SIPS), 2011 IEEE Workshop on, IEEE, 2011, pp. 81–86.
[15] E. A. Lee, D. G. Messerschmitt, Static scheduling of synchronous data flow programs for digital signal processing, IEEE Trans. Comput. 36 (1) (1987) 24–35.
[16] E. A. Lee, D. G. Messerschmitt, Synchronous data flow, Proceedings of the IEEE 75 (9) (1987) 1235–1245.
[17] M. Wipliez, Compilation infrastructure for dataflow programs, Ph.D. thesis, INSA Rennes (2010).
[18] G. Bilsen, M. Engels, R. Lauwereins, J. Peperstraete, Cyclo-Static Dataflow, IEEE transactions on signal processing 44 (2) (1996) 397–408.
[19] M. Wipliez, M. Raulet, Classification and transformation of dynamic dataflow programs, in: Design and Architectures for Signal and Image Processing (DASIP), 2010 Conference on, IEEE, 2010, pp. 303–310.
[20] M. Pelcat, J. Piat, M. Wipliez, S. Aridhi, J. Nezan, An open framework for rapid prototyping of signal processing applications, EURASIP Journal on Embedded Systems 2009 (2009) 3.
[21] B. Bhattacharya, S. S. Bhattacharyya, S. Member, Parameterized Dataflow Modeling for DSP Systems, IEEE Transactions on Signal Processing 49 (2001) 2408–2421.
[22] J. T. Buck, Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model, Ph.D. thesis, EECS Department, University of California, Berkeley (1993).
[23] J. Boutellier, C. Lucarz, S. Lafond, V. Gomez, M. Mattavelli, Quasi-static scheduling of CAL actor networks for reconfigurable video coding, Journal of Signal Processing Systems.
[24] J. Gorin, F. Preteux, M. Raulet, Optimized dynamic compilation of dataflow representations for multimedia applications, To appear in Special Issue on Networked digital media, Annals of Telecommunications.
[25] I. Richardson, S. Kannangara, M. Bystrom, J. Philp, M. de Frutos Lopez, A framework for fully configurable video coding, in: Picture Coding Symposium, 2009. PCS 2009, IEEE, 2009, pp. 1–4.
[26] J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, Y. Xiong, Taming heterogeneity-the Ptolemy approach, Proceedings of the IEEE 91 (1) (2003) 127–144.
[27] R. Esser, J. Janneck, A framework for defining domain-specific visual languages, Workshop on Domain Specific Visual Languages, ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA-2001), 2001.
[28] S. Bhattacharyya, G. Brebner, J. W. Janneck, J. Eker, C. von Platen, M. Mattavelli, M. Raulet, OpenDF: a dataflow toolset for reconfigurable hardware and multicore systems, SIGARCH Comput. Archit. News 36 (5) (2008) 29–35.
[29] Xilinx DSP Division, XLIM: An XML Language-Independent Model, ASTG technical memo, D2c (September 2007).
[30] J. W. Janneck, I. Miller, D. Parlour, G. Roquier, M. Wipliez, M. Raulet, Synthesizing Hardware from Dataflow Programs: An MPEG-4 Simple Profile Decoder Case Study, Journal of Signal Processing Systems 63 (2) (2011) 241–249. doi:10.1007/s11265-009-0397-5.
[31] C. von Platen, CAL ARM Compiler, Tech. rep., D2c (2008-2011).
[32] P. Ward, The transformation schema: An extension of the data flow diagram to represent control and timing, Tech. rep., Yourdon, Inc., New York, NY 10036 (1986).
[33] J. Whaley, Joeq: A virtual machine and compiler infrastructure, in: Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators, ACM, 2003, pp. 58–66.

[34] P. Le Guernic, A. Benveniste, P. Bournai, T. Gautier, Signal–a data flow-oriented language for signal processing, Acoustics, Speech and Signal Processing, IEEE Transactions on 34 (2) (1986) 362–374.

[35] C. Lattner, V. Adve, The LLVM Instruction Set and Compilation Strategy, Tech. Report UIUCDCS-R-2002-2292, CS Dept., Univ. of Illinois at Urbana-Champaign (Aug 2002).

[36] C. Lattner, V. Adve, Llvm: A compilation framework for lifelong program analysis & transformation, in: Code Generation and Optimization, 2004. CGO 2004. International Symposium on, IEEE, 2004, pp. 75–86.

[37] J. Le Feuvre, C. Concolato, J. Moissinac, GPAC: open source multimedia framework, in: Proceedings of the 15th international conference on Multimedia, ACM, 2007, pp. 1009–1012.

[38] J. Gorin, M. Wipliez, F. Preteux, M. Raulet, A portable Video Tool Library for MPEG Reconfigurable Video Coding using LLVM representation, in: Design and Architectures for Signal and Image Processing (DASIP), 2010 Conference on, IEEE, 2010, pp. 183–190.

[39] J. Gorin, M. Wipliez, F. Prêteux, M. Raulet, LLVM-based and scalable MPEG-RVC decoder, Journal of Real-Time Image Processing 6 (1) (2011) 59–70.

[40] J. Gorin, M. Raulet, Y. L. Cheng, H. Y. Lin, N. Siret, K. Sugimoto, G. G. Lee, An RVC dataflow description of the AVC Constrained Baseline Profile decoder, in: Image Processing (ICIP), 2009 16th IEEE International Conference on, 2010, pp. 753–756.