RVC: A MULTI-DECODER CAL COMPOSER TOOL

F. Palumbo, D. Pani, E. Manca, L. Raffo *

DIEE - Dept. of Electrical and Electronic Eng.
University of Cagliari
09123 Cagliari, Italy

M. Mattavelli, G. Roquier

EPFL
CH-1015, Lausanne
Switzerland

ABSTRACT

The Reconfigurable Video Coding (RVC) framework is a recent ISO standard aiming at providing a unified specification of MPEG video technology in the form of a library of components. The word "reconfigurable" evokes run-time instantiation of different decoders starting from an on-the-fly analysis of the input bitstream.

In this paper we move a first step towards the definition of systematic procedures that, based on the MPEG RVC specification formalism, are able to produce multi-decoder platforms, capable of fast switching between different configurations. Looking at the similarities between the decoding algorithms to implement, the papers describes an automatic tool for their composition into a single configurable multi-decoder built of all the required modules, and able to reuse the shared components so as to reduce the overall footprint (either from a hardware or software perspective). The proposed approach, implemented in C++ leveraging on Flex and Bison code generation tools, typically exploited in the compilers front-end, demonstrates to be successful in the composition of two different decoders MPEG-4 Part 2 (SP): serial and parallel.

Index Terms— RVC, RVC-CAL language, MPEG standard, reconfigurable systems

1. INTRODUCTION

The MPEG Reconfigurable Video Coding (RVC) framework is a recent ISO standard defining the methodology and formalism for the specification of video coding technology. The main innovation is based on the adoption of a dataflow model of computation expressed using the RVC-CAL language and the associated network language FNL whose definitions are part of the new MPEG standard [1]. Compared to the traditional ways of providing specifications based on textual descriptions and on C/C++ monolithic reference software implementations, the new standard provides the specification of video codecs in forms of dataflow programs composed by a network of Functional Units (FUs) belonging to a standard Video Tool Library (VTL). The specification of a codec by means of this formalism, which is a form of executable program providing a full functional validation, leads to an *abstract* codec specification, that can be translated into a proprietary implementation by replacing the FUs of the "abstract" network with custom hw/sw implementations of the corresponding components. Such composability property of dataflow specifications is a very attractive feature enabling to conceive implementations procedures based on static or dynamic reconfigurations.

A possible exploitation of the MPEG RVC framework is the definition of hardware reconfigurable multi-standard video decoders able to configure at run-time the correct decoder for the incoming video bitstream, as depicted in Fig. 1. A consequence of the modular approach of an RVC specification is that several FUs are common among different codecs, so that reusing the available elements in the VTL becomes a natural implementation solution.

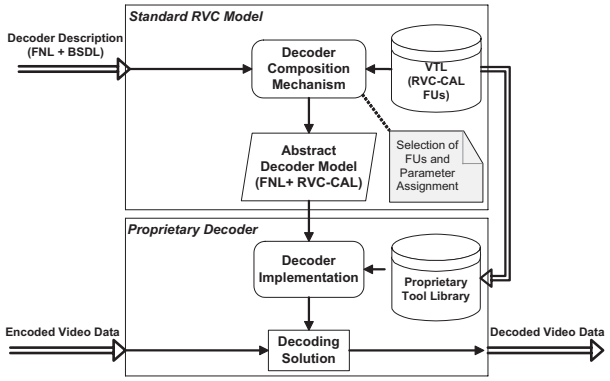

Fig. 1. The RVC design model.

In this paper we present a C++ tool, called Multi-Decoder CAL Composer (MDCC), for the automatic merging of multiple decoder descriptions with the purpose of creating a single multi-standard decoder instance that can be easily converted into a proprietary hw/sw implementation. As a matter of fact,

*The research leading to these results has received funding from the European Communitys Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 248424, MADNESS Project, and by the Region of Sardinia, Young Researchers Grant, PO Sardegna FSE 2007-2013, L.R.7/2007 "Promotion of the scientific research and technological innovation in Sardinia"

when several decoders described by using the RVC standard approach need to be implemented, a manual procedure to accomplish this task would be very prone to errors and resource consuming. The proposed tool analyzes the different RVC standard "abstract decoder specifications" and inserts configurable Switching Boxes (Sboxes) to enable non-shared modules to be used only by the specific decoder configuration that require them. In this way the final implementation does not require a complete decoder instantiation to meet the requirements of the incoming video bitstream, but rather a reconfiguration of the multi-standard implementation in order to activate only the needed FUs. Such a general approach could be exploited for both hardware and software implementations, reducing the time required to switch between different decoders. Similar approaches have been proposed in different fields, such as the one presented in [2] for multimode DSPs. The MDCC tool leverages on Flex and Bison code generation tools, typically exploited for the compilers front-end, in order to parse the abstract description of two decoders to create two C++ graphs. These graphs are then merged into a single composed graph, recursively extended in order to take into account more decoder descriptions. The final output is also made available as a network of FUs in order to comply with the standard RVC approach.

The rest of this paper is organized as follows. In Sect. 2 information about the RVC and the available tools are given. In Sect. 3 the proposed approach is described whereas the results on a case study (a dual-standard decoder) are discussed in Sect. 4. Section 5 concludes.

2. BACKGROUND ON MPEG RVC

RVC is a new MPEG standard [1, 3, 4, 5], but unlike previous standards, it does not define a new codec, but rather a framework for the description of new codecs [6]. Following the principles of the design for reusability, RVC fosters the fast development of new video codecs through either the combination of different FUs in non-standard ways or via the extension of the VTL with new algorithms (which undergo a standardization process at library component level) for the description of new video codecs.

The framework is based on the RVC-CAL language [1], a dataflow-oriented actor language based on the *actor* entity, which is an abstract representation of a computing element working asynchronously in order to generate output data sequences from input ones. Inputs and outputs take the form of *tokens* and specific tokens configurations can fire an action inside the actor, possibly leading to the emission of output tokens as a final outcome of a state change of the actor. Since actors do not expose their internal state, all the inter-actors interactions can take place only by means of tokens exchange. Within the RVC framework, actors are used to describe the FUs that constitute the VTL. RVC-CAL is a standard subset of the original CAL language introducing some restrictions aimed at the easier development of automatic hw/sw generation tools. Actor interconnections in forms of networks create intrinsically parallel dataflow descriptions which are at the basis of the codec descriptions, involving also token-passing FIFOs in order to provide appropriate buffering for the inter-actor communications [7]. The RVC standard explicitly introduces 3 framework elements [8]:

- VTL [4]: a library of FUs (RVC-CAL actors) for video coding;
- FNL [3]: the FU Network Language, an XML-based language for the description of the actor networks as $.nl$ files;
- BSDL [5]: the MPEG-21 Bitstream Syntax Description Language (also XML-based) to describe the behaviour of the bitstream parser to be implemented in the decoder to properly analyze the encoded stream.

At the time being, several tools have been created to support developers to migrate towards the new RVC standard specification formalism. OpenDF [9] is an interpreter infrastructure for the simulation of hierarchical networks of actors. Leveraging on the adoption of a library of common FUs, RVC also enables the development of automatic tools to map the VTL onto a library of hw/sw modules to be used for the real implementation of the decoder (Fig. 1). At now, two compiler infrastructures are ready for the automatic code generation from RVC-CAL descriptions for both hardware (FPGA) and software (multicore processors) architectures. On the hardware side, Openforge is a supporting tool that turns CAL models into hardware description languages (VHDL/Verilog) [10]. On the software side, the Open RVC-CAL Compiler (ORCC) [11] is a supporting tool that turns CAL models into software implementations in several programming languages (C/C++, LLVM, etc.) [12].

3. MULTI-DECODER CAL COMPOSER

The MDCC tool has been conceived in order to provide a unique description of a multi-decoder system, preserving the functionalities of the different integrated dataflows, to be used by both hardware architects and software developers. The former will be able to assemble a platform to implement the behaviour of different decoders on the same system, with fast switching between different algorithms. The latter will have available a comprehensive description of the integrated decoders within a single code.

MDCC works inserting, where needed, an Sbox in the FNL description of the multi-decoder, under the hypothesis that the considered decoders have some parts in common that can be shared in a multi-decoder environment. As Fig. 2 shows, the final output of the MDCC tool is the *directed graph*

(DG) of the overall multi-decoder environment, while the inputs are the N different CAL dataflow descriptions of the decoders to be integrated. We assume that these decoders are provided in terms of RVC-CAL atomic actors and their interconnections in terms of FNL (*.nl* files).

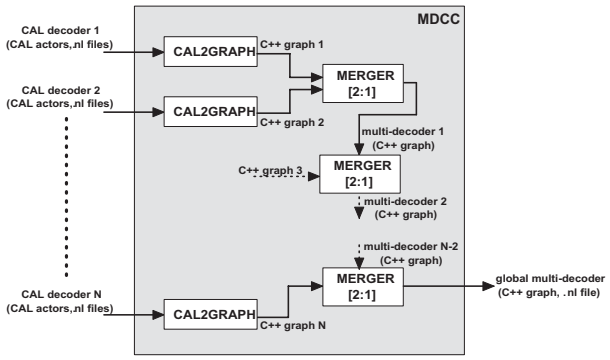

Fig. 2. Multi-Decoder CAL Composer (MDCC).

Figure 2 highlights also which are the two main components of the MDCC tool:

- the *CAL2graph interpreter* creates the C++ *DG* of a single CAL dataflow;
- the *Merger* algorithm creates the multi-decoder C++ *DG* starting from two *DG*s.

3.1. The CAL2graph interpreter

The *CAL2graph interpreter* operates on the single decoder basis, creating the *DG* of a single CAL decoder. Its inputs are: the FNL description of a single decoder with all the atomic actors and/or the other *.nl* files describing sub-networks of actors. At the end of the *DG* creation process, the output graph will be pruned of any node corresponding to a network of actors by means of a flattening operation, able to transform the original *DG* in a graph composed by just atomic CAL actor nodes. In the C++ *DG*, the nodes are representative of the atomic actors whereas the arcs of the interconnections among them. The *CAL2graph interpreter* works in three steps:

- the analysis of the *.nl* file through a lexical analyzer that generates the Lexical Tokens (LTs) to be processed by the grammar interpreter;
- the analysis of the LTs through the grammar interpreter that determines the *DG* using the grammar rules associated to the different sequences of LTs;
- the recursive iteration of the first two steps until the *DG* is pruned by not atomic actors.

The first two phases of the *CAL2graph interpreter* have been realized using Flex and Bison [13], which are two code-generating tools expressively designed to aid in compiler development. Flex, which stands for fast lexical analyzer, is able to take a sequence of characters and to break them apart into LTs. These LTs are then processed by Bison, which is able to associate actions to pre-defined sequences of LTs. These sequences are normally known as a *grammar*.

In the *CAL2graph interpreter*, Flex has been used to realize the lexical analyzer and Bison to define the grammar interpreter. The Bison parser combines the sequences of LTs as received by the lexical analyzer and, according to the matched grammar rule, performs different actions to create the *DG* of the single decoder. Mainly, as highlighted in Fig. 3, there are three categories of rules implemented by the Bison grammar interpreter, which are related to:

- nodes management (*Actor Interface Section* in Fig. 3);
- connections management (*Link Section* in Fig. 3);
- recursion management (*Final Section* in Fig. 3).

In practice the *CAL2graph interpreter*, in each stage of its recursion, first of all instantiates the nodes of the *DG* (*Actor Interface Section* in Fig. 3) and then creates the oriented arcs among them (*Link Section* in Fig. 3). It starts parsing the top *.nl* file of the decoder and, as soon as an actor or another *.nl* file (correspondent to a CAL sub-network) is found, a node of the *DG* is constructed. For all the non atomic actors, a reference is pushed in a queue in order to apply recursion on them. In the *Link Section* of Fig. 3, the arcs between two nodes are created and the undertaken actions are different according to the received sequence of LTs:

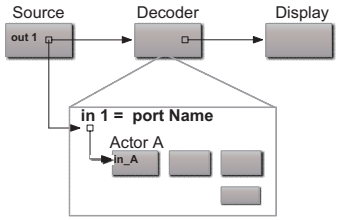

Fig. 4. Connection of the input port of a sub-network to the actor output port of the original *DG*.

- *portname→actorname.portname* implies to connect:
  - one of the network input ports with the proper actor input port (top $.nl$ file analysis)
  - the input port of the sub-network with the actor output port in the upper level $.nl$ file (generic recursive step, see Fig. 4);
- *actorname1.portname→actorname2.portname* implies to connect two intermediate actors;
- *actorname.portname→portname* implies to connect:
  - one of the network output ports with the proper actor output port (top $.nl$ file analysis)

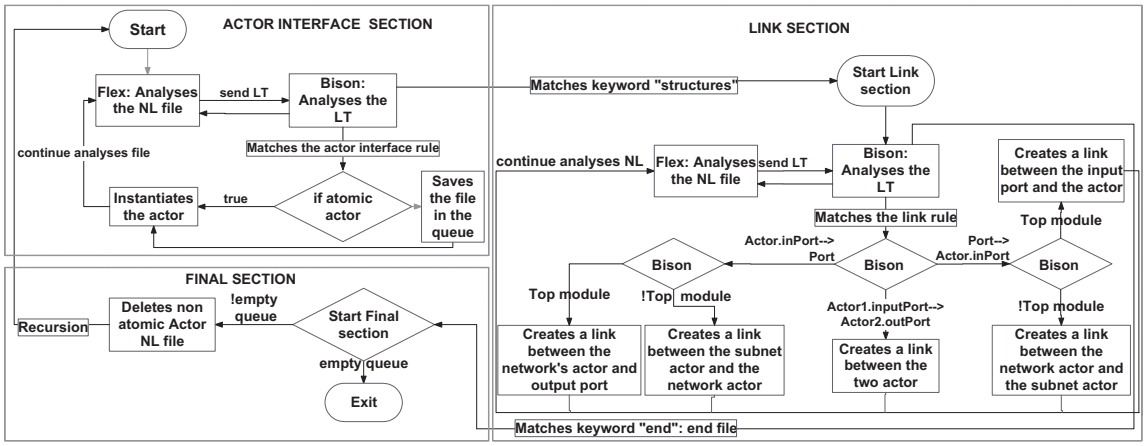

Fig. 3. The *CAL2graph interpreter*.

– the output port of the sub-network just constructed with the actor input port in the upper level *.nl* file (generic recursive step, see Fig. 5);

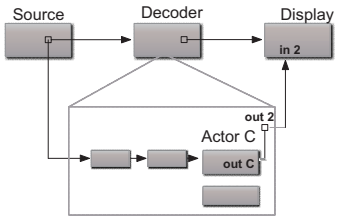

Fig. 5. Connection of the output port of a sub-network to the actor input port of the original *DG*.

As soon as all the nodes have been instantiated and their connections established, the *Final Section* (Fig. 3) can take place: the *CAL2graph interpreter* pops a reference to a *.nl* sub-network from the queue (if any) and executes all the previously described steps again. At the end of each recursion, the newly created sub-graph is substituted to the non atomic node placed originally in the global *DG* (see Fig. 6). As already said, the *CAL2graph interpreter* will not stop until the *DG* is not composed of atomic actors only, meaning that the *.nl* references queue is empty.

3.2. The Merger algorithm

The *Merger* algorithm is responsible of taking two atomic *DG*s and combining them, in order to build the multi-decoder *DG*. At the first step, as it is clear in Fig. 2, it combines the *DG* of two decoders. Then, until there are decoders to be integrated in the multi-decoder structure, it combines a *DG* of a single decoder with the multi-decoder *DG* created in the previous run. Then, the final *DG* will be representative of the complete reconfigurable multi-decoder. In this part of the MDCC tool the Sbox unit is introduced.

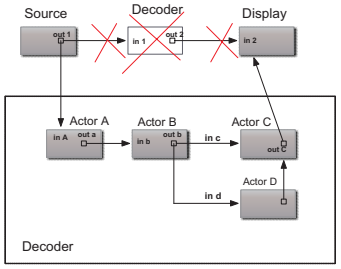

Fig. 6. Substitution of a non atomic node with its correspondent sub-network.

3.2.1. The Switching Box Unit (Sbox)

The Sbox in the multi-decoder offers high-speed runtime reconfiguration capabilities. Mainly the Sbox allows to change, at runtime, the topology of the connections among the actors of the different integrated dataflows in order to define which one of the integrated decoders is in use. It is conceived to be placed at the crossroads between different paths of the multi-decoder to allow more than one dataflow to share a common dataflow where some actors are in common and some others not. It is clear that this element will play a key role when the output provided by the MDCC tool to the harware architects will be translated into a real hardware platform. In this case the Sbox units must be able to be programmed:

- to connect each one of their input ports to the output ports, in order to change at run-time the topology;
- to define at run-time some of its possible internal parameters that can differ from one decoder to another, such as the width (in bits) of the input and output connections, the pipeline stages placed at the input and at the output ports or the width (in bits) of the ports.

The *Merger* algorithm instantiates the Sbox units as simple graph nodes. It will be responsibility of the MDCC tool

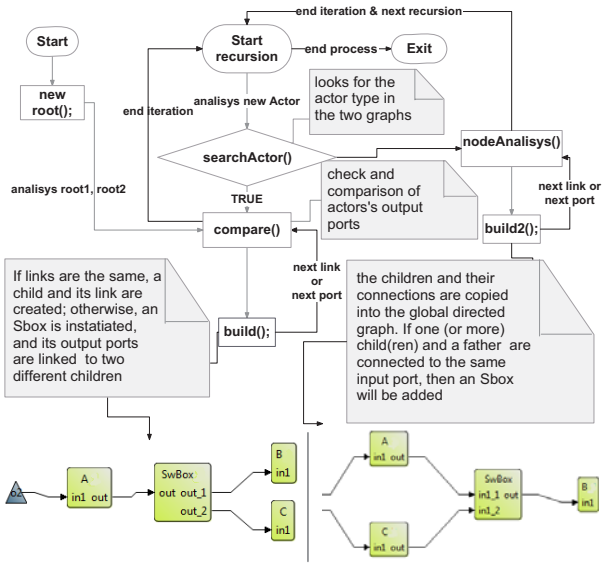

Fig. 7. The *Merger* algorithm.

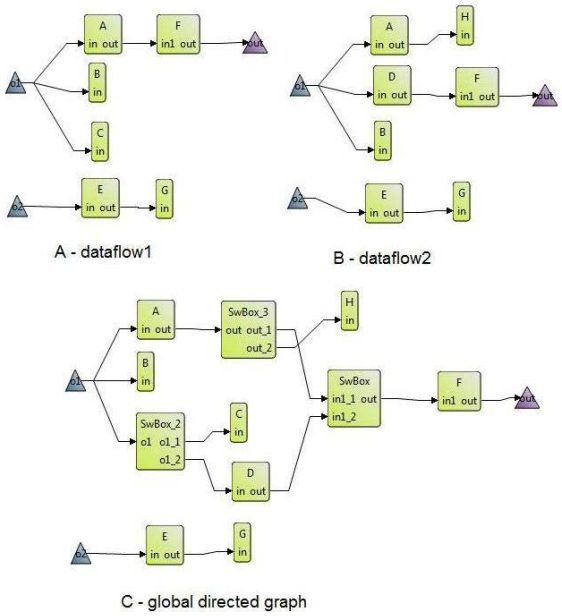

Fig. 8. Single dataflow 1 (A), single dataflow 2 (B) and correspondent global *DG* (C). All these graphs have been plotted using Graphiti.

users to decide how to use these elements for the final multi-decoder implementation.

3.2.2. The Algorithm

Figure 7 depicts the flow chart of the *Merger* algorithm. It starts analyzing the graphs from their roots, which are the global inputs of the system. The algorithm starts considering the first input of the system. At first, it checks where it is connected in the single *DG*s to be merged (*compare()* function). Then, it verifies (*build()* function) if the interconnections among the considered input and the actors input port in the two *DG*s to be merged are the same or not to instantiate accordingly the proper nodes and interconnections in the global *DG*. The *Merger* algorithm iterates these steps for any interconnection of the roots, then recursion can take place.

In the recursive steps, the algorithm considers, one at a time, the nodes already instantiated in the global *DG* and not processed yet. It checks whether the considered node is present in both the *DG*s to be merged or not (*searchActor()* function). If it is so, the *Merger* algorithm iterates the *compare()* and *build()* functions on all its output ports to combine the single *DG*s together. If it is not, it iterates the *nodeAnalysis()* and *build2()* functions to preserve the dataflow of the single decoders in the multi-decoder environment. The *Merger* algorithm will continue recursion until all the children nodes of the already analyzed nodes are not processed.

3.2.3. Example of the Algorithm Application

Here follows a description of a step by step application of the *Merger* algorithm; clearly we are not considering here any real application but just two simple dataflows. *o1* and *o2* are the roots of the system. The *Merger* algorithm iterates three times on *o1* comparing, in the two *DG*s, its interconnections. In the two *DG*s of the dataflow1 (Fig. 8.A) and the dataflow2 (Fig. 8.B) the connections between *o1* and the actors A and B are found to be the same by the *compare()* function; therefore in the global *DG* they will be copied as they are by the *build()* function. That is not the case for the third iteration on *o1*, which is connected to the actor C in the dataflow 1 and to the actor D in the dataflow 2. The *build()* function in this case has to insert a Sbox node between *o1* and the C and D nodes; this Sbox will have an input arc connected to *o1* and two output arcs (*o1_1* and *o1_2*) connected respectively to C and D. The same steps are applied to *o2*, which connection is simply replicated in the global *DG* since it is the same in both dataflows. The recursion on the already instantiated children nodes (A, B, C, D and E) can start.

Actor A is found to be present in both graphs by the *searchActor()* function. It has only one output arc then just one iteration of the *compare()* and *build()* functions is required: a Sbox node is being inserted to redirect data towards H and F. Recursion on actor B, even though it is present in both dataflows, does not generate any iteration of the algorithm since node B has no output arc. The same applies for the C node. As soon as node D is processed, the *Merger* algorithm recognizes that, even though D is present just in the dataflow2 (according to the *searchActor()* function output), its child F is already instantiated and connected in the global *DG*, therefore the *build2()* function instantiates a Sbox node

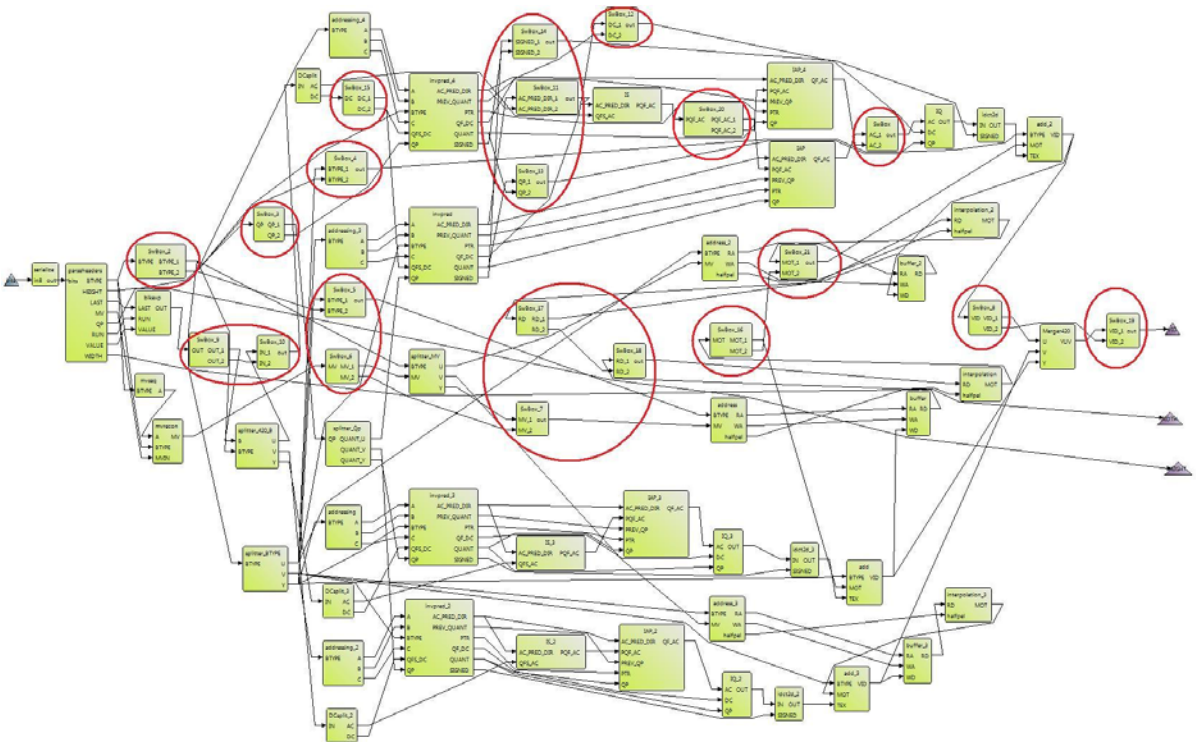

Fig. 9. The multi-decoder graph obtained merging the parallel MPEG-4 SP decoder and the serial MPEG-4 SP decoder single graphs using the *Merger* algorithm.

and modifies the already established connection to allow both the previous path and D to connect to the same F input port. The actor E is present in both dataflows (according to the *searchActor()* function output) and the *compare()* and *build()* functions has to simply replicate G and its interconnection in the global *DG* (being the same in both dataflows).

At this point the *Merger* algorithm has accomplished the analysis of the children of the roots, but still G, F and H have to be processed. The analysis of the G node proceeds as the B and C ones and no iterations of the algorithm are required. The same applies for H. F is present in both dataflows (according to the *searchActor()* function output) and it is found to be connected to the same global output port (according to the *compare()* and *build()* functions output), which is therefore replicated in the global *DG* using the *build()* function.

4. RESULTS

This Section discusses a use case of the MDCC tool and its possible exploitations. More space is accorded to the possible hardware applications, since one of the future development of this research is the completion of the MDCC tool with an automatic hardware platform generator tool. Even though, the MDCC tool can be also used from a software development perspective.

4.1. The parallel MPEG-4 SP and the serial MPEG-4 SP Use Case

In a first time, the MDCC tool has been tested over simple factitious dataflows, similar to the ones presented in the example discussed in Sect. 3.2.3. In all those cases it was possible to exploit the Graphiti tool to verify the outputs both of the *CAL2graph interpreter* and of the *Merger*. In order to do that, both components of the MDCC tool provide as output also the *.nl* file of the C++ graph they are generating, to be easily displayed using Graphiti.

In terms of real applications, two different decoders have been processed by the MDCC tool: the parallel MPEG-4 SP decoder and the serial MPEG-4 SP decoder. Using Graphiti we have compared the graphs of the parallel MPEG-4 SP decoder and of the serial MPEG-4 SP decoder generated using the *CAL2graph interpreter* with respect to the ones of the decoders prior to apply the MDCC tool. In both cases the graphs before and after the application of the MDCC tool were the same, taking into account the flattening process.

Moreover it was also possible to operate on the processed decoders a functional test by means of OpenDF. We substituted, in the pre-processed CAL decoders, the original FNL

description of the decoder with the flattened one produced by the *CAL2graph interpreter*. It resulted that it was possible to operate anyway a complete video decoding, meaning that the decoder functionalities were not compromised at all by the flattening operation performed by the *CAL2graph interpreter*.

The case study adopting the serial MPEG-4 SP decoder and the parallel MPEG-4 SP decoder has been carried out merging their single *DG*s into the multi-decoder *DG* applying the *Merger* algorithm. The resulting multi-decoder *DG* is depicted in Fig. 9, the inserted Sbox units highlighted using circles. The effectiveness of the MDCC approach is confirmed by Table1, which compares the number of actors to be integrated on two separated decoders (last two rows of the *Actor Sum* column) and those of the merged multi-decoder *DG* (last two rows of the *MDCC applied* column). Considering the *TOT actors (no Sbox)* it is clear that the multi-decoder *DG* integrates less actors (46 instead of 59). Moreover, even considering the presence of the Sbox units (*TOT actors (with Sbox)* row), it should be considered that it is less costly to integrate Sbox units (which in hardware are nothing more than programmable multiplexers) than more complex actors such as an interpolation unit or a complete DCT. Clearly the more decoders will be merged the more will be the instantiated Sbox units.

Without a CAL implementation of the Sbox unit, the *.nl* file of the multi-decoder cannot be simulated using OpenDF. Nevertheless, the achieved multi-decoder *DG* is a fundamental result to be provided to hardware architects and software developers:

- providing an idea of the common parts two decoders are able to share;
- enabling an easier management of a multi-decoder environment;
- enabling the possibility of extracting some possible metrics of the final multi-decoder platform directly at this level, before the final implementation.

4.2. The MDCC tool exploitation

One of the future developments of the MDCC tool is its possible exploitation at a lower level of abstraction to create a physical multi-decoder platform. We are already working on the possibility of completing the MDCC with a tool called *graph2HW*. The main objective of the *graph2HW* tool will be the automatic creation of the HDL description of the overall multi-decoder platform taking as inputs:

- the multi-decoder *DG* provided by the MDCC tool;
- the HDL actors library composed of all HDL descriptions of the FUs involved in the multi-decoder platform (possibly using OpenForge);
- the programmable HDL description of the Sbox.

Table 1. Comparison of the adopted resources merging or not the parallel MPEG-4 SP decoder and the serial MPEG-4 SP decoder.

Actor	Parallel SP	Serial SP	Actor Sum	MDCC Applied
serialize	1	1	2	1
parseheaders	1	1	2	1
blkexp	1	1	2	1
mvseq	1	1	2	1
mvrecon	1	1	2	1
DCsplit	3	1	4	3
splitter_420_B	1	0	1	1
splitter_MV	1	0	1	1
spliter_Qp	1	0	1	1
splitter_BTYPE	1	0	1	1
DCRaddressing_16x16	1	0	1	1
Algo_DCRaddressing_8x8	2	0	2	2
Algo_DCRaddressing	0	1	1	1
Algo_DCRinvpred_luma_16x16	1	0	1	1
Algo_DCRinvpred_chroma_8x8	2	0	2	2
Algo_DCRinvpred	0	1	1	1
IS	3	1	4	3
Algo_IAP_16x16	1	0	1	1
Algo_IAP	0	1	1	1
Algo_IAP_8x8	2	0	2	2
IQ	3	1	4	3
idct2d	3	1	4	3
add	3	1	4	3
address	3	1	4	3
buffer	3	1	4	3
interpolation	3	1	4	3
Merger420	1	0	1	1
Sbox	0	0	0	17
TOT actors (no Sbox)	43	16	59	46
TOT actors (with Sbox)	43	16	59	63

The C++ *DG* of the multi-decoder environment provided by the MDCC tool stores already all the necessary information to create the multi-decoder platform top module, since:

- each node stores the name, the type and the value of each possible parameter of the related HDL module, necessary for its instantiation;
- each arc is representative of an interconnection among the FUs or with the external environment.

Assuming that the input bitstream to be processed in a real multi-decoder hardware implementation is able to carry the information relative to the type of processing to be performed, it will be possible to configure at run-time the Sbox units and all the FU parameters present in the architecture. At the moment we are assuming that the input bitstream will carry a sort of ID to identify the datapath to be implemented among a pre-defined set. We are exploring the possibility of providing dynamic reconfigurability exploiting a smart agent to be integrated in the multi-decoder evironment able to create the proper configuration starting from meta-information in the input bitstream and a complete library of FUs.

The multi-decoder *DG*, as already mentioned in Sect. 4.1, can also provide important profiling evaluations related to the

final decoder, a lot before the final hardware implementation. This profiling can provide some directions for better performing the reconfiguration at the hardware level, but it can also guide software developers in the applications conception. To do that, the multi-decoder *DG* has to be back-annotated with some extra information, e.g.:

- the activation percentage of each FU (obtained through high-level execution profiling),
- the relevance of each FU (e.g. related to the number of decoders it belongs to and to its frequency of use),
- the latency and the physical metrics of each FU (area occupation and maximum frequency obtained running off-line some single decoder hardware simulation).

All these information can aid hardware architects and software developers when some trade-off choices have to be taken. For example, on the basis of the back-annotated graph, hardware architects aiming at minimizing resources utilization can face the decision whether or not adopting partial reconfiguration of the multi-decoder platform and they can also determine the parts of the platform to be partially reconfigured being less used than others. Moreover, since the MDCC tool is able to create the global *DG* of any multi-decoder platform in a few minutes, it is also possible to exploit it in order to decide whether it is convenient or not to create a real multi-decoder environment. It might happen in fact that the effort needed is wasted since the different decoders do not share enough actors or the back-annotated graph of the multi-decoder does not show enough costs saving.

5. CONCLUSIONS

This paper presented the Multi-Decoder CAL Composer tool: a tool able to define a multi-decoder structure starting from the FNL specification of the single decoders as defined by the RVC standard. This tool is based on the adoption of the Flex and Bison code generation tools and on C++ in order to analyze the different RVC decoders and to merge them together through the adoption of some switching units, i.e. the Sbox units. The multi-decoder produced by the MDCC tool is composed of: FUs commons to different decoders, which are shared among them in the multi-decoder platform, and other FUs that are used just when needed by the sub-sets of decoders they originally belong to. The strength of this approach is the possibility of easily switching from one decoder to another on the fly during system execution, whether it is hardware or software implemented, avoiding a complete context switch. The developed tool has also been integrated with other state of the art CAL tools such as Graphiti and OpenDF and, even though specifically designed to address video decoding applications, it will be possible to use it to compose any CAL dataflows.

6. REFERENCES

[1] M. Mattavelli, I. Amer, and M. Raulet, "The reconfigurable video coding standard," *Signal Processing Magazine, IEEE*, vol. 27, no. 3, pp. 159–167, May 2010.

[2] Vinu Vijay Kumar and John Lach, "Highly flexible multimode digital signal processing systems using adaptable components and controllers," *EURASIP J. Appl. Signal Process.*, vol. 2006, january.

[3] "ISO/IEC 23001-4 (2009). MPEG systems tech.—Part 4: Codec configuration representation," .

[4] "ISO/IEC 23002-4 (2010). MPEG video tech.—Part 4: Video tool library," .

[5] "ISO/IEC 23001-5 (2008): MPEG systems tech.—Part 5: Bitstream syntax description language (BSDL)," .

[6] Christophe Lucarz, Ihab Amer, and Marco Mattavelli, "Reconfigurable Video Coding : Objectives and Technologies," in *IEEE Intl. Conf. on Image Processing, Cairo, Egypt*, 2009.

[7] I. Amer, C. Lucarz, G. Roquier, M. Mattavelli, M. Raulet, J.-F. Nezan, and O. Deforges, "Reconfigurable video coding on multicore," *Signal Processing Magazine, IEEE*, vol. 26, no. 6, pp. 113 –123, nov. 2009.

[8] Shuvra S. Bhattacharyya, Johan Eker, Jorn Janneck, Christophe Lucarz, Marco Mattavelli, and M. Raulet, "Overview of the MPEG Reconfigurable Video Coding Framework," *Jour. of Signal Processing Systems*, 2009.

[9] Shuvra S. Bhattacharyya, Gordon Brebner, Jörn W. Janneck, Johan Eker, Carl von Platen, Marco Mattavelli, and Mickaël Raulet, "OpenDF: a dataflow toolset for reconfigurable hardware and multicore systems," *SIGARCH Comput. Archit. News*, vol. 36, no. 5, pp. 29–35, 2008.

[10] Jorn W. Janneck, Ian D. Miller, David B. Parlour, Ghislain Roquier, Matthieu Wipliez, and Mickael Raulet, "Synthesizing Hardware from Dataflow Programs," *Journ. of Signal Processing Systems*, 2009.

[11] The Open RVC-CAL Compiler, ," `http://orcc.sourceforge.net/`.

[12] Matthieu. Wipliez, Ghislain Roquier, and Jean-François Nezan, "Software code generation for the rvc-cal language," *Journal of Signal Processing Systems*, 2009.

[13] John Levine, Tony Mason, and Doug Brown, *lex & yacc, 2nd Ed. (A Nutshell Handbook)*, O'Reilly, 1992.