Code generation for the MPEG Reconfigurable Video Coding framework: From CAL actions to C functions

Matthieu Wipliez, Ghislain Roquier, Mickael Raulet, Jean François Nezan, Olivier Déforges

▶ **To cite this version:**

Matthieu Wipliez, Ghislain Roquier, Mickael Raulet, Jean François Nezan, Olivier Déforges. Code generation for the MPEG Reconfigurable Video Coding framework: From CAL actions to C functions. Multimedia and Expo, (ICME) 2008 IEEE International Conference on, Jun 2008, Hannover, Germany. pp.1049 - 1052, 10.1109/ICME.2008.4607618 . hal-00336487

HAL Id: hal-00336487
https://hal.science/hal-00336487

Submitted on 4 Nov 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

CODE GENERATION FOR THE MPEG RECONFIGURABLE VIDEO CODING FRAMEWORK: FROM CAL ACTIONS TO C FUNCTIONS

Matthieu Wipliez, Ghislain Roquier, Mickaël Raulet, Jean-François Nezan, Olivier Déforges

IETR laboratory, UMR CNRS 6164, Image and Remote Sensing Group
INSA de Rennes, 20 Avenue des Buttes de Coësmes, 35043 RENNES Cedex, FRANCE
Contacts: {mwipliez, groquier, mraulet, jnezan, odeforge}@insa-rennes.fr

ABSTRACT

The MPEG Reconfigurable Video Coding (RVC) framework aims to provide a unified specification of all video technology. In the RVC framework, a decoder is build in a modular manner as a configuration of video tools taken from the MPEG toolbox library. The elements of the library are specified using the CAL Actor Language (CAL). CAL is a data flow based language providing computation models that are concurrent and modular. This paper describes a synthesis tool that from a CAL specification generates an executable SW module. Code generators are fundamental supports for the deployment and success of the MPEG RVC framework. This paper focuses on the automatic translation of a CAL action, which is the first step to a complete actor translation. The techniques described here is capable of automatically generating C executable code according to a finite set of rules. This approach has been used to obtain a C implementation of the IDCT module which is one element of the RVC library. The generated code is validated versus the original CAL description and simulated using the Open Dataflow environment.

***Index Terms*—** MPEG RVC, Caltrop Actor Language

1. INTRODUCTION

A large number of successful MPEG video coding standards have been developed since the first MPEG-1 standard finalized in 1988. The standardization process has always aimed to provide appropriate forms of specifications for a wide and easy deployment. While at the beginning MPEG-1 and MPEG-2 were only specified by textual descriptions, starting with MPEG-4 C/C++ descriptions, called reference software, became the formal specification of the standard and the textual part became an additional component with the aim of clarifing the reference SW descriptions. Such descriptions are composed of non-optimized software packages and face now many serious limitations. For real implementations all reference SW have to be re-written from the beginning to optimize performances and to adapt it to current design flow methodologies. Such monolithic specifications hide the inherent parallelism and data flow structure of the video coding algorithms, features that are necessary to be exploited for efficient implementations. In the meanwhile the growth of video coding technologies leads to solutions that are increasingly complex to be designed and present significant overlap between standards. Moreover, several codecs and associated profile levels have to coexist in a single product. This is the result of the standardization process: adding tools in a standard involves a new process in which all the technology is modified. In short, the process has to be improved to face its limitations and to avoid bottlenecks in the future.

The observation of these drawbacks of current video standard specification formalism led to the development of the Reconfiguration Video Coding (RVC) standard. The key concept was to design a decoder at high system level to avoid low-level implementation considerations during the system specification stage. An ”abstract” model focusing on functionality and concurrency is the specification formalism chosen, which is the best starting point for any design and implementation process. RVC provides a high-level description of the MPEG standard using a specific language called CAL. Once the high level model/specification is available the challenge is then to develop appropriate tools providing optimized implementations. Indeed, efficient SW/HW implementations need hardware and software code generators. Some work have be done to directly generate HDL from a CAL model [1]. The work presented here aims to automatically generate optimized software for multiprocessor and multicore targets. To achieve these ambitious objectives the first step is to provide a non-normative software synthesis tool called Cal2C. It is designed to become a tool available in the RVC framework. The main issue is to deal with various syntaxes allowed in the CAL language. This paper proposes a safe process to handle CAL operators, analyze them and generate functional C code. The paper is organized as follows: sections 2 and 3 introduce the RVC framework. This is followed by a description of Cal2C functionalities. As part of the MPEG-4 SP decoder, the IDCT case study is reported in section 5. Finally conclusions and several perspectives are given in section 6.

2. RVC FRAMEWORK

2.1. Scope

The MPEG RVC framework is currently under development by MPEG as the part of MPEG-B and MPEG-C standards. It aims to provide a new interoperable model of defining MPEG standards at system-level [?]. An abstract decoder is built in a block diagram manner in which blocks define processing entities called Functional Units (FUs) and connections represent the data path. RVC provides both a normative standard library of FUs and a set of decoder descriptions as a network of FUs. Such a representation is modular and helps the reconfiguration of a decoder by modifying the topology of the network or adding new FUs to describe new standards. RVC mainly focuses on reusability by allowing decoder descriptions to contain common FUs across standards.

2.2. FU specification

The Cal Actor Language (CAL) has been chosen as the normative language to specify the standard library of FUs. CAL is a suitable language for the modular approach of RVC. CAL is a dataflow oriented language created as part of the Ptolemy project [4]. A CAL actor is a computational entity with interfaces (input and output ports), internal state and parameters. An actor is strongly encapsulated; an actor can neither access nor modify the state of any other actor. An actor may only interact with others by sending data (called tokens) along communication channels. When an actor is executed (fired), it consumes tokens from input ports, changes its internal state and produces tokens on the output ports. An actor is executed in a sequence of steps called actions. An action is a piece of computation that an actor performs during firing. An action is specified by the number of tokens it consumes and produces. An actor may contain any number of actions. When an actor is fired, it has to select one of them based on the availability of input tokens and optionally based on conditions relating to their values or the current state. In addition, it may also contain constructs that constrain action selection. An action guard enables conditional action firing according to input tokens or state variables. A finite state machine (FSM) allows actions to be scheduled according to the current state of the actor and action priorities enabled to order action selection. In short, an action is fireable if it respects the following conditions:

- √ there are sufficient tokens on input ports and adequate room on output ports;
- √ guard clauses evaluate to true;
- √ the current state enables the action to fire according to the FSM;
- √ no higher-priority action may fire.

2.3. RVC Decoder Description

On the other hand, the Decoder Description Language (DDL) enables the description of the decoder structure. A decoder is a network formed by a set of interconnected actors. DDL is an XML-based language that can be used to specify the topology of the network. DDL is hierarchical - a network may be a part of a more general network - and is used to pass parameters to actors. Decoder descriptions may be generated automatically by a graphical composition tool called Moses[1] that is a CAL editing and simulation framework. The Open Dataflow project[2] is an environment for building and simulating CAL actor models with a textual description of the network (using the Network Language) that may be automatically translated into an RVC decoder description. This non-normative tool also provides an automatic HDL synthesis from a CAL network [1].

3. CODE GENERATION PRELIMINARIES

3.1. Concurrent programming

A dataflow network is inherently concurrent. Understanding concurrency brings with it understanding of how the whole system behaves and how interactions between actors are done. Scheduling policy and data transfer mechanism are the key concepts when interpreting a dataflow network. CAL does not have any bias towards any particular network interpretation. In other words, a CAL network exhibits neither the scheduling policy nor the data transfer mechanism. Formal models are used to interpret a network. This paper does not intend to draw up an exhaustive list of models: cf. [5] for more details. However, two particular models are chosen to demonstrate dataflow network interpretation. On one hand, the Process Network (PN) model [6] is efficient when describing asynchronous systems. The PN model is determinate: the output result is not affected by the scheduling algorithm. The downside of this model is the asynchronous execution. It requires run-time scheduling, which makes the system verification harder. On the other hand, the synchronous dataflow (SDF) model has more constraints but is suitable for fixed-rate systems [7]. It is especially well-adapted for software synthesis; a network analysis may lead to compile-time scheduling. This being so, indications about real-time behavior or execution in bounded memory can be determined at compile-time. However, the fixed-rate requirement makes it impossible to model more flexible systems like those in video coding. Indeed, CAL actors in the standard library often contain multiple actions and an abundance of priorities, FSM or guard clauses. It leads to state-dependent and/or conditional execution that makes it difficult to analyze the network. Then, the PN model is a more appropriate model if implementing such networks.

[1] http://www.tik.ee.ethz.ch/ moses/
[2] http://opendf.sourceforge.net/

3.2. Software synthesis

The goal of the C code generator is to produce a code that behaves exactly like the CAL reference code. Non-automatic verification is a manual comparison between the generated and the original files. This requires the source code produced to be recognizable. Another approach to the checking of program behavior, required for a decoder to be qualified as RVC-compliant, is to assert that the tokens consumed and produced by each actor equal the tokens described by the norm. In both cases, it is preferable to generate a C code that has a similar structure to the CAL code: an actor is translated into a file, an action into a function. There are different means of translating CAL to C while meeting the above requirements. The first is to use source transformation programming systems, such as Stratego/XT or TXL. Such systems are very general, and based on context-free grammars and rules. Their purpose is to perform transformation from a language to another. Another possibility, since we are specifically aiming at producing C code, is to use C-specific transformation tools. SUIF (Stanford University Intermediate Format) and CIL (C Intermediate Language [8]) can parse C code to an Intermediate Representation (IR), perform transformations on it, and print it back to C. These systems have the advantages of producing a clean representation and a pretty-printer, without having to learn another language or have a complete toolset. We chose CIL over SUIF because its IR has clean semantics, as detailed in section 4.3, and produces a higher quality C code.

4. FROM CAL ACTIONS TO C FUNCTIONS

Translating the actions of a CAL actor to C functions consists of the following steps: a CAL file is parsed to an abstract tree representation of the source program (section 4.1); this tree is then annotated with type information (section 4.2), and converted to CIL Intermediate Representation (section 4.3). Finally, CIL pretty-prints the IR to C code.

4.1. CAL parser

The CAL parser parses a CAL file using an LALR parser created from the rules described in the Caltrop Language Report [4], and produces an Abstract Syntax Tree (AST). In this tree, a variable become a Var node, an access to the ith element of a list, an I node, unary and binary operations, U and B nodes respectively. Assignments, function applications, and conditionals are transformed in Assign, App, and If nodes. The AST holds the same semantic information as the source code, but its form makes it easier to process automatically.

4.2. Type inference

The AST obtained at this stage may not have complete type information, as CAL allows the programmer to omit type annotations of declared variables whereas C does not. This makes it necessary to guess types when they are not explicit: this step is called type inference. To this end, we use the type system and the $\mathcal{W}$ algorithm defined by Damas and Milner in [**?**]. This algorithm is able to infer a type-scheme for any expression in the λ-calculus formalism, extended with the *let* clause. A λ-expression is either a term, a function application, a function definition, or a local definition:

$$e \;:=\; x \mid e_1 e_2 \mid \lambda x.e \mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2$$

One property of this formalism is that it can be used to express any computable function. To compute the type of CAL expressions, we transform them to their equivalent λ-calculus representation using the rules shown in Table 1. Arithmetic operations become λ-terms with known types: unary and binary operators become functions with one and two arguments respectively. An access to a list element and a conditional become function applications. The nth function takes two parameters of types α list and integer, and returns an element of type α. The "if" function accepts a boolean parameter, two parameters of type α, and returns a result of type α.

CAL abstract syntax	**λ-calculus**
Var(v)	v
I($list$, e)	$(nth\ list)\ e$
U(op, e)	op e
B(e_1, op, e_2)	(op e_1) e_2
App($e, e_1, e_2, \ldots, e_{n-1}, e_n$)	$(\ldots((e\ e_1)\ e_2)\ldots e_{n-1})\ e_n$
If($e_{cond}, e_{then}, e_{else}$)	((**if** e_{cond}) e_{then}) e_{else}
Lambda($[p_1,\ p_2,\ \ldots,\ p_n]$, $[v_1 = e_1, \ldots, v_n = e_n]$, e)	$\lambda p_1.\lambda p_2.\ldots\lambda p_n.$ **let** v_1 = e_1 **in** $\ldots$ **let** v_n = e_n **in** e

Table 1. Conversion rules

After conversion, we apply the $\mathcal{W}$ algorithm to the λ-expressions associated with each declared variable. If there is a type error, the algorithm fails, and the process stops. Otherwise, the variables are updated with the type information obtained.

4.3. Converting to CIL

C expressions can be purely functional or make use of side-effects: functions are allowed to modify global state or local arguments. CIL goes further by distinguishing control flow statements and instructions among side-effect expressions, from functional side-effect free expressions. Control flow statements include branch statements (goto, while. etc.) and conditionals. Instructions are assignments and function calls. Expressions include constants, unary and binary operations, and the operator address. Lvalues, that can appear at the left of an assignment, are expressed in terms of host

and offset, and have precise typing rules. The conversion of CAL AST to CIL is done by a translation function hereinafter called T. This function takes a CAL expression, and returns a CIL expression and a list of statements, with no particular difference between control flow statements and instructions. The environment Γ maps variables names to their corresponding CIL lvalues. The $newVar$ function creates a new temporary variable named β, and returns $Lval(\beta)$. The expression resulting from $T(e)$ is shown below:

$$\begin{aligned}
Var\ var &\rightarrow Lval(\Gamma(var)) \\
B(e_1, op, e_2) &\rightarrow BinOp(op, T(e_1), T(e_2)) \\
U(op, e) &\rightarrow UnOp(op, T(e)) \\
I(host, e) &\rightarrow Lval(host, Index(T(e), 0)) \\
Assign(e_{dst}, e_{src}) &\rightarrow T(e_{dst}) \\
\left.\begin{array}{l} App(e_{fun}, e_{args}) \\ If(e_{cond}, e_{then}, e_{else}) \end{array}\right\} &\rightarrow newVar()
\end{aligned}$$

The statement list returned for side-effect free expressions is empty. An assignment returns a Set instruction, a function application and a Call instruction with result placed in β while a conditional returns a statement of the form:

If(e_c, e_t, e_e)	$(e_c,\ s_c) = T(e_c), (e_t,\ s_t) = T(e_t)$ $(e_e,\ s_e) = T(e_e)$ $[s_c;\ If(T(e_c), [s_t;\ \beta = e_t], [s_e;\ \beta = e_e])]$

5. RESULTS

The CAL-based MPEG4-SP standard is used to validate the C code generation. Yet, about 90% of actions in the decoder are successfully translated with CAL2C (the remaining 10% actions are located in the bitstream parser subnetwork [**?**] which require few other investigations). In order to test the functionality, the case study application is the two-dimensional IDCT, which is part of the MPEG4-SP decoder. CAL networks are implemented using the PN model; the POSIX threads are used to implement actors and circular buffers are used for communication. Actions are automatically generated by CAL2C and are subroutines handled by the actor thread. Two IDCT are used [**?**, 10]. The first one is depicted in Figure 1, composed of 2 input ports, 1 output port and 5 actors. They have almost the same scheme, separately applying one-dimensional IDCT along the row and column axis of an 8×8 pixel block. The testbench consists of applying stimulus (a random number generator) to the top network and verifying the response against an expected result (from the original CAL description simulated using the Open Dataflow environment). Many scenarios have been tested during which no errors occur that confirm the validity of CAL2C. Note that the compiling is done faster in this approach and the use of C enhances IDCT run times.

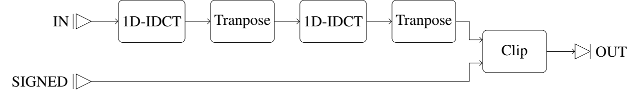

Fig. 1. 2D-IDCT CAL network

6. CONCLUSION AND FUTURE WORK

This paper has presented a brief overview of the software used to translate CAL actions to C functions. The process has been successfully applied to translate the normative IDCT, as well as most of the actions of RVC MPEG-4 SP decoder actors. However, FSM, action priorities and guard clauses are not yet considered. The automatic actor code generation must be completed by considering the relevant control structures of the CAL language. Moreover, network implementation (which has been a manual task until now) has to be automatically generated from the top network. Finally, the ultimate goal of the software synthesis in RVC is to provide a tool that enables to generate code from CAL actors as well as schedule and map the network onto SW architectures. To this end, network analysis and CAL coding practice rules may lead to compile-time scheduling of MPEG standards.

REFERENCES

[1] Jorn Janneck, "From actors to gates," in *CHESS seminar*, October 2007, UC Berkeley.

[2] C. Lucarz, M. Mattavelli, J. Thomas-Kerr, and J. Janneck, "Reconfigurable media coding: a new specification model for multimedia coders," in *Proceedings of SIPS'07*, Oct. 2007.

[3] J. Eker and J. Janneck, "Cal language report," Tech. Rep. ERL Technical Memo UCB/ERL M03/48, University of California at Berkeley, Dec. 2003.

[4] J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendor, e Sonia, and S. Yuhong, "Taming heterogeneity—the ptolemy approach," in *Proceedings of the IEEE*, Jan. 2003, vol. 91.

[5] G. Kahn, "The semantics of a simple language for parallel programming," in *Proceedings of IFIP '74*, Aug. 1974, pp. 471–475.

[6] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. Comput.*, vol. 36, no. 1, pp. 24–35, 1987.

[7] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, "CIL: An Infrastructure for C Program Analysis and Transformation," in *Proceedings of CC 2002*, Apr. 2002, pp. 213–228.

[8] Luis Damas and Robin Milner, "Principal type-schemes for functional programs," in *Proceedings of POPL '82*, 1982, pp. 207–212.

[9] J. Thomas-Kerr, J. Janneck, M. Mattavelli, I. Burnett, and C. Ritz, "Reconfigurable media coding: self-describing multimedia bitstreams," in *Proceedings of SIPS'07*, Oct. 2007.

[10] "IEEE standard specifications for the implementations of 8x8 inverse discrete cosine transform," *IEEE Std 1180-1990*, Mar. 1991.

[11] ISO / IEC 23002-2, "Fixed-point 8x8 IDCT and DCT," 2007.