Automatic software synthesis of dataflow program: An MPEG-4 simple profile decoder case study
Ghislain Roquier, Matthieu Wipliez, Mickael Raulet, Jörn W. Janneck, Ian D. Miller, David B. Parlour

To cite this version:

HAL Id: hal-00336516
https://hal.archives-ouvertes.fr/hal-00336516
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.
AUTOMATIC SOFTWARE SYNTHESIS OF DATAFLOW PROGRAM: AN MPEG-4 SIMPLE PROFILE DECODER CASE STUDY

Ghislain Roquier, Matthieu Wiplier, Mickaël Raulet

IETR/INSIA, UMR CNRS 6164
Image and Remote Sensing laboratory
F-35043 Rennes, France
{groquier, mwipliez, mraulet}@insa-rennes.fr

Jörn W. Janneck, Ian D. Miller, David B. Parlour

Xilinx inc.
San Jose, CA, U.S.A
{jorn.janneck,ian.milller,dave.parlour}@xilinx.com

ABSTRACT

The MPEG Reconfigurable Video Coding (RVC) framework is a new standard under development by MPEG that aims at providing a unified high-level specification of current MPEG video coding technologies. In this framework, a decoder is built as a configuration of video coding modules taken from the standard “MPEG toolbox library”. The elements of the library are specified by a textual description that expresses the I/O behavior of each module and by a reference software written using the CAL Actor Language. A decoder configuration is written in an XML dialect by connecting a set of CAL modules. Code generators are fundamental supports that enable the direct transformation of a high level specification to efficient hardware and software implementations. This paper presents a synthesis tool that from a CAL dataflow program generates C code and an associated SystemC model. Experimental results of the RVC Expert’s MPEG-4 Simple Profile decoder synthesis are reported. The generated code and the associated SystemC model are validated against the original CAL description which is simulated using the Open Dataflow environment.

Index Terms— MPEG RVC, CAL Actor Language, dataflow modeling, software synthesis

1. INTRODUCTION

A large number of successful MPEG video coding standards have been developed since the first MPEG-1 standard in 1988. The standardization process has always aimed at providing 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, reference software became the formal specification of the standard. Such descriptions are composed of non-optimized software packages and face now many limitations. These monolithic specifications hide the inherent parallelism and the dataflow structure of the video coding algorithms. For efficient implementations the reference software has to be rewritten to exploit this so-called parallelism. Meanwhile, the growth of video coding technologies leads to solutions that are increasingly complex and more difficult to design. So far, no effort has been made to profit from the significant overlap between standards.

The observation of these drawbacks of current video standard specifications led to the development of the Reconfigurable Video Coding (RVC) standard. The key concept behind this project is to enable the design of decoders at a higher level of abstraction. An “abstract” model focusing on modularity, concurrency and reusability is a better starting point for any design and implementation process. RVC provides a high-level description of the MPEG standard using as new form of reference software, for each module of the standard library, a specific language called CAL. Once the high-level model is available the challenge is then to develop appropriate tools that implement the design flow and provide the optimization steps necessary for efficient implementations. This paper presents a non-normative (in terms of relation with the RVC standard) software synthesis tool called Cal2C that from a dataflow program generate C code and associated SystemC model in a completely automated process.

The paper is organized as follows: section 2 introduces the RVC framework. This is followed by the description of the CAL actor simulator and synthesis tools for RVC in section 3. Synthesis process of the CAL to C transformation are next explained in section 4. Results on the RVC expert’s MPEG-4 SP decoder are reported in section 5. Finally conclusions and future work are provided in section 6.

2. RVC FRAMEWORK

The MPEG RVC framework is currently under development by MPEG as part of MPEG-B and MPEG-C standards. RVC aims at providing a model of specifying existing or completely new MPEG standards at system-level 11. An abstract decoder is built as a block diagram 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 ex-
pressed as networks of FUs. Such a representation is modular and helps the reconfiguration of a decoder by modifying the topology of the network. RVC mainly focuses on reusability by allowing decoder descriptions to contain common FUs across standards.

2.1. FU specification

The CAL Actor Language [2] has been chosen as the language of the reference software that specify the normative I/O behaviors of the modules of the RVC standard library of FUs. A CAL actor is a computational entity with interfaces (input and output ports), internal state and parameters. An actor is strongly encapsulated; it can neither access nor modify the state of any other actor. An actor may only interact with others by sending data (called tokens) along channels. During an execution actions (also referred as firing an action), it can consume input tokens, produce output tokens and change its internal state. Each action specifies the number of tokens it consumes and produces. When an action is fired, it has been selected based on the availability of input tokens and optionally based on additional conditions (called action guards) which may depend on token values or the current actor state. Action selection may be further constrained using action schedules as a Finite State Machine (FSM). Action firing is thus state-dependent. Finally, action priorities can be used to impose a partial order among the actions to be selected, in case more than one action is otherwise enabled. In short, an action is fireable if it respects the following conditions:

1 – there are the necessary tokens on input ports;
2 – guard clauses, if present, evaluate to true;
3 – the current state enables the action to fire according to the FSM;
4 – no higher-priority action respects (1), (2) and (3).

2.2. RVC Decoder Description

An RVC decoder is described with the Decoder Description Language (DDL), an XML dialect that enables the description of the decoder connectivity. A decoder is a network formed by a set of interconnected actors. DDL is hierarchical - a network may be a part of a more general network - and is used to pass parameters to actors. For instance, the graphical representation of the macroblock-based MPEG-4 Simple Profile decoder description is shown Figure 1. The parser and the inverse DCT (Figure 2) blocks are hierarchical networks of actors (each of them described in a separate DDL file). They are represented with a shadowed block. All other blocks are atomic actors specified in CAL. Note that for readability, only one edge is represented when two actors are connected by more than one edge.

3. SUPPORT TOOLS FOR RVC

3.1. Simulator

CAL is supported by a portable interpreter infrastructure that can simulate a hierarchical network of actors. This interpreter was first used in the Moses project. Moses features a graphical network editor, and allows the user to monitor actor execution (actor state and token values). The project being no longer maintained, it has been superseded by the Open Dataflow environment (OpenDF for short). Contrarily to Moses, this project does not provide a network graphical editor. Networks have been traditionally described in a textual language called Network Language (NL), which can be automatically converted to DDL and vice versa. It is also possible to use the Graphiti editor to display networks in the

---

3.2. Software synthesis

It is important to be able to automatically obtain a concrete software implementation from a dataflow description. The C language is particularly well-suited as a target language. The same code can be compiled on any processor, from embedded DSPs and ARMs to general-purpose microprocessors, which considerably eases the task of writing a software synthesis tool. The interest of having an automatic C software synthesis is two-folded. The code obtained can be executed, in which case it enables a considerably faster simulation of the dataflow program and the ability to debug the program using existing IDEs (Visual Studio, Eclipse CDT). Moreover, the C code description obtained may be a basis for an optimized tailor-made decoder. For these reasons, we created the Cal2C tool (detailed in section 4) that aims at producing functionally-equivalent, humanly-readable C code from CAL descriptions.

3.3. Hardware synthesis

Another tool is available that converts CAL to HDL [3]. After parsing, CAL actors are instantiated with the actual values for their formal parameters. The result is an XML representation of the actor which is then precompiled (transformation and analysis steps, including constant propagation, type inference and type checking, analysis of data flow through variables...), represented as a sequential program in static single assignment (SSA) form (making explicit the data dependencies between parts of the program).

Then follows the synthesis stage, which turns the SSA threads into a web of circuits built from a set of basic operators (arithmetic, logic, flow control, memory accesses and the like). The synthesis stage can also be given directives driving the unrolling of loops, or the insertion of registers to improve the maximal clock rate of the generated circuit.

The final result is a Verilog file containing the circuit implementing the actor, and exposing asynchronous handshake-style interfaces for each of its ports. These can be connected either back-to-back or using FIFO buffers into complete systems. The FIFO buffers can be synchronous or asynchronous, making it easy to support multi-clock-domain dataflow designs.

4. CAL2C SOFTWARE SYNTHESIS

4.1. Semantics of CAL dataflow

The system behavior of a dataflow program is determined by the interactions between actors (i.e. exchange of data tokens). Such interactions are governed by a Model of Computation (MoC) that defines which scheduling policies can be used to fire actors. The CAL language is not related to any particular dataflow MoCs. Indeed, several forms of dataflow exist to interpret the network from the general dataflow process network (PN) [4] model with multiple firing rules to the more restrictive synchronous dataflow (SDF) one [5, 6]. CAL extends the model in [4] by:

1 – state within actors,
2 – multiple overlapping (non-joinable in the parlance of [4]) firing rules, and
3 – priorities among firing rules.

CAL actors in the standard library often contain multiple actions and priorities, FSM or guards that lead to state-dependent or conditional execution. Therefore most of the CAL actors in the library are closer to the PN model which is then chosen (as a first milestone) for developing the tool described in this paper.

When the PN model is chosen to interpret CAL networks, any environment which supports multithreading may be chosen for implementation. For instance, it can be done using POSIX threads by translating CAL actors into threads and by replacing connections with FIFOs. The PN model can also be executed by a discrete-event (DE) scheduler, such as the one provided by SystemC. Actors and FIFOs can be easily translated using SystemC classes. Additionally, its simulation environment permits high-level programming, well-adapted to functional verifications. A PN-oriented SystemC model is expressed as a network of modules communicating with each other via blocking FIFOs (with an additional way to support
if different names to avoid linking problems, even if they are
lated code. For example, (1) distinct functions should have
functions, procedures and actions are translated. The C lan-

4.2.1. Actor code translation

Finally an additional file is created to instantiate the top net-
tworks are instantiated and connected. CAL actor transla-
ted into a header file where FIFOs, modules and sub-
several files as explained in Figure 3. Each NL network is
software synthesis from a network of CAL actors produces
over, this work is a first step to an efficient “pure C” code gen-
convenient PN modeling and simulation environment. More-

Fig. 4. C translation of a CAL action
token availability). Note that Cal2C does not intend to use SystemC as an hardware description language but only as a
convenient PN modeling and simulation environment. More-
over, this work is a first step to an efficient “pure C” code gen-
eration (closer to an SDF model than a PN one). In short, the
software synthesis from a network of CAL actors produces
several files as explained in Figure 3. Each NL network is
translated into a header file where FIFOs, modules and sub-
networks are instantiated and connected. CAL actor transla-
done in 2 different parts: the actor code (actions, functions, procedures) to express the functionality and the action scheduler (priorities, FSMs, guards) to control the execution. Finally an additional file is created to instantiate the top network and to launch SystemC simulation.

4.2. C code generation: Actors

4.2.1. Actor code translation

Translating the actor code produces a single C file wherein
functions, procedures and actions are translated. The C lan-
guage and compilers impose some limitations on the trans-
lated code. For example, (1) distinct functions should have
different names to avoid linking problems, even if they are

in different actors. Another challenge is (2) the difference
of programming paradigm between the source and the target
language: CAL allows functional constructs that have no di-
rect equivalent in C. The action translation process starts with
an Abstract Syntax Tree (AST) issued from the CAL source
code, and modifies it as needed to meet the previous require-
ments. Function names are prefixed with the actor name to
prevent any potential name clashes; actor parameters are re-
placed by their values (when constant) or transformed to local
variables otherwise; actions are converted to functions where
input and output patterns become parameters. Finally, actor
declarations are ordered by dependencies between locals, so
that a variable or a function is defined before being used. At
this point, we convert the AST to λ-calculus, apply Damas-
Michlη \( \lambda \)-algorithm \[7\] to it, and augment the AST with the
type information returned. Types are necessary for correct C
code generation, and type-dependent transformations: we in-
line functions that return lists, and compute list sizes. The
transformed CAL AST is expressed in the C Intermediate
Language (CIL) \[8\], where CAL functional constructs are re-
placed by imperative ones. C code is generated by calling the
pretty-printer included in the framework.

The actor code translation process is illustrated on figure 4. Figure 4(a) is an action from the Inversequant actor of the
RVC reference MPEG4-SP decoder, and figure 4(b) is the
translated C code. The resulting code exhibits a function whose
name is composed of the actor name and the action
name (requirement (1)). Its parameters are the same as the
action’s, with an additional pointer to a structure containing
the actor variables. The if expression has been transformed
to assignments of temporary variables \( \text{if}_7, \text{if}_8 \) (require-
ment (2)). As a matter of fact, function calls have also become
assignments of temporary variables \( \text{call}_6, \text{call}_9 \) because
CIL semantics requires it. The action output expression is
translated as a pointer parameter whose value is written at
the end of the C function. The synthesized C code shown in
figure 4(b) is functionally-equivalent to the CAL code and re-
mains humanly-readable, which were two requirements listed
in section 3.2.

4.2.2. Action scheduling

An action scheduler is created to control the action selection
during execution. Priorities, guards, token consumption rates
and FSM have to be translated to this end. Determining the
overall order of action execution is required to have a con-
sistent evaluation of actions that can be fired. Priorities are
resolved by sorting actions in a total order and by adding a
if-then-else statement around actions wherein the condition
is given by the availability of input tokens and the guards
conditions. FSMs are resolved using switch-case statement.
Finally, the generated file consists of a thread with an infi-
nite loop wherein its body consists of the result of the previ-
ous transformations and actions are replaced with their corre-
sponding C functions.

For instance, a downsampler by N is illustrated figure 5(a). It could be written in a simpler manner but this actor enables to highlight key features of the action scheduling translation. The synthesized C++ code is illustrated figure 5(b).

```plaintext
actor downsampler (N) In ⇒ Out:
  count := 1;
  pass:
    action In: [x] ⇒ Out: [x] end
done:
  action ⇒ guard count = N do count := 1; end
  skip:
    action In: [x] ⇒ do count := count + 1; end
  schedule fsm pass:
    copy ( pass ) --> discard;
    discard ( done ) --> copy;
    discard ( skip ) --> discard;
  priority
    done > skip;
end
```

(a) CAL downsampler

```plaintext
struct downsampler_vars {
  int count;
  int N;
};
void downsampler::process() {
  int fsm_state, _call_6, _call_7, _out_1;
  struct downsamplerN_vars _actor_vars;
  _actor_variables.count = 1;
  fsm_state = 1;
  while (1) {
    switch (fsm_state) {
      case 1:
        _call_6 = In->get();
        downsampler_pass(&_actor_vars, _call_6, &_out_1);
        Out->put(_out_1);
        fsm_state = 2;
        break;
      case 2:
        if (_actor_vars.count == _actor_vars.N) {
          downsampler_done(&_actor_vars);
          fsm_state = 1;
        } else {
          _call_7 = In->get();
          downsampler_skip(&_actor_vars, _call_7);
          fsm_state = 2;
        }
        break;
    }
  }
}
```

(b) Synthesized action scheduler

Fig. 5. Action scheduling (FSM) of CAL actor

### 4.3. SystemC code generation: Networks

Expressing a NL network in SystemC is relatively straightforward: actor or network instantiations are transformed to module instantiations. A SystemC module is declared using the `SC_MODULE` macro. Its effect is that it creates a C++ class that can be used inside a SystemC simulation. For convenience, in Cal2C each module is translated to a definition file (*.h) and an implementation file (*.cpp). There are two semantic differences between the SystemC implementation and the NL specification: FIFO channels, while implicit in NL, must be explicitly created in SystemC. Broadcasting data from a source to several sinks is transparent in NL, but requires additional logic in SystemC, namely a generic broadcast module.

### 5. RESULTS ON THE MPEG-4 SIMPLE PROFILE DECODER

In order to certify the correctness of Cal2C code translation, the first case study is a two-dimensional inverse DCT. The IDCT is a component of several MPEG standard decoders and is specified by the new Finite Precision IDCT Specification [9] based on [10]. The algorithm consists of applying one-dimensional IDCT along the row and column axis of an $8 \times 8$ pixel block. The network is composed of 2 input ports, 1 output port and 5 different actors; one actor can be instantiated several times in NL. Incoming tokens from IN port are dequantized data and a token from SIGNED enables to specify to the ‘‘clip’’ actor if incoming data are signed or not. The first row of Table 1 shows the number of files of the respective programs. An actor becomes a C, a C++ and a header file; a network simply becomes a header file; the additional C++ file is the main. The second row exhibits the corresponding source lines of code (SLOC). The testbed consists of applying a stimulus (streamed by a C-code reference software of MPEG4 SP decoder) to the top network and verifying the response against an expected result (from the CAL description simulated compared to a “golden reference” streamed by the C-code reference software).

<table>
<thead>
<tr>
<th>IDCT</th>
<th>CAL</th>
<th>NL</th>
<th>C</th>
<th>C++</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of files</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Code Size (SLOC)</td>
<td>131</td>
<td>25</td>
<td>324</td>
<td>386</td>
<td>107</td>
</tr>
</tbody>
</table>

Table 1. Code size and number of files automatically generated for the IDCT

Another synthesized model of a more complex CAL dataflow program simulated with the Open Dataflow environment also validate the Cal2C tool. The compilation process has been successfully applied to the full MPEG-4 Simple Profile dataflow program written by the MPEG RVC experts (Figure 1). Table 2 shows that synthesized C-software is faster than the simulated CAL dataflow program (20 frames/s instead of 0.15 frames/s), and close to real-time for a QCIF format (25 frames/s). However it remains slower than the automatically synthesized hardware description by Cal2HDL [3]. Using Cal2C has also permitted to correct some actors which had a behavior depending on one particular code generator. Indeed, they made use of nondeterminism allowed by CAL (for instance, 2 fireable actions without priority relationship). The way to solve the nondeterminism (not expressible in C) is implementation-dependent.
The MPEG4 SP decoder program is composed of 27 atomic actors; atomic actors can be instantiated several times, for instance there are 42 actor instantiations in this dataflow program. Its number of SLOC is shown in Table 3. All of the generated files are successfully compiled by gcc. For instance, the “ParserHeader” actor inside the “Parser” network is the most complex actor with multiple actions. The translated C-file (with actions and state variables) includes 1043 SLOC for actions and 1895 for action scheduling. The original CAL file contains 962 lines of codes as a comparison.

<table>
<thead>
<tr>
<th>MPEG4 SP decoder</th>
<th>Speed</th>
<th>Code size</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAL simulator</td>
<td>0.015</td>
<td>3.4</td>
</tr>
<tr>
<td>Cal2C</td>
<td>2</td>
<td>10.4</td>
</tr>
<tr>
<td>Cal2HDL</td>
<td>290</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 2. MPEG4SP decoder speed and SLOC

6. CONCLUSION AND FUTURE WORKS

This paper presents a software synthesis tool that enables an automatic translation of dataflow programs written in CAL, showing the rules to translate the I/O behaviors (actions, functions and procedures), the relevant control structures (priority, guard and FSM) as well as the networks. Moreover, the process has been successfully applied to automatically translate the MPEG-4 SP decoder. The results obtained so far show the efficiency and soundness of the synthesised code.

The next milestone of Cal2C concerns the static scheduling of such dataflow programs. Indeed, its main purpose is to generate efficient embedded software for multiprocessor target. Even if the PN interpretation of the network gives relevant results for hardware, it is no longer adapted for software when it comes to hard real-time implementation. Liveness and memory boundedness are critical issues that cannot be guaranteed. More restrictive dataflow models (like SDF models) enable static scheduling with efficient code generations. However, only a subset of PN-actors can be scheduled statically (especially in the RVC standard library). The next step is to identify static regions that obey SDF restrictions. Indeed, it is sometimes straightforward to schedule actors (for instance an actor with one action and no guard is SDF). Unfortunately, several actors in the RVC library are not so trivial. Future work will have to provide analysis of such actors and/or CAL coding guidelines in order to achieve static scheduling of (parts of) standard decoders.

Once this work is done, it may be possible to generate both hardware and software for implementation onto heterogeneous platforms with processors and programmable logic devices. Moreover, we are in the process of integrating Cal2C within the Open Dataflow framework, which should make various heterogeneous implementations easier to obtain.

7. REFERENCES