J Sign Process Syst
DOI 10.1007/s11265-009-0390-z

Software Code Generation for the RVC-CAL Language

**Matthieu Wipliez · Ghislain Roquier ·
Jean-François Nezan**

Received: 23 January 2009 / Revised: 20 April 2009 / Accepted: 8 June 2009

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 and future MPEG video coding technologies using dataflow models. In this framework, a decoder is built as a configuration of video coding modules taken from the standard MPEG toolbox library or proprietary libraries. 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 a subset of the CAL Actor Language named RVC-CAL. 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. The generated code is validated against the original CAL description simulated using the Open Dataflow environment. Experimental results of the translation of two descriptions of an MPEG-4 Simple Profile decoder with different granularities are shown and discussed.

M. Wipliez (✉) · J.-F. Nezan
IETR/INSA Rennes, 35043, Rennes, France
e-mail: Matthieu.Wipliez@insa-rennes.fr

J.-F. Nezan
e-mail: Jean-Francois.Nezan@insa-rennes.fr

G. Roquier
EPFL, 1015 Lausanne, Switzerland
e-mail: Ghislain.Roquier@epfl.ch

Keywords Reconfigurable Video Coding · CAL actor language · Dataflow programming · Software code generation

1 Introduction

Processor frequencies no longer double every two years as predicted by Moore's law. Multicore architectures are now being proposed so that computational power keeps increasing. The main issue of this approach is that standard programming languages (C, C++, ...) are sequential and not suitable for parallel multicore architectures. The dataflow programming concept consists of describing an application with a directed graph where edges represent the flow of data between operations. Dataflow programming highlights the potential parallelism of the application, which can be used to distribute calculations over available cores.

MPEG is currently working on the development of the RVC standard. The key concept behind this project is to describe decoders using dataflow graphs: RVC provides a high-level description of the MPEG standard written in a specific language called RVC-CAL. Appropriate tools must implement the design flow and provide the optimization steps necessary for efficient implementations. Uniprocessor software code generation must be addressed before considering more complex implementations (multicore or hardware/software co-design). This paper presents a non-normative (in terms of relation with the RVC standard) software code generator called Cal2C that from a dataflow program generates C code and an associated SystemC model. We show that the algorithm described with RVC-CAL

Published online: 27 June 2009

can be automatically transformed, compiled and executed efficiently on a uniprocessor system.

The paper is organized as follows: Section 2 presents dataflow programming within the RVC-CAL environment. Section 3 describes the Cal2C software code generator. Results obtained with Cal2C are shown in Section 4. Section 5 outlines perspectives of future work, while we conclude in Section 6.

2 Dataflow Programming for RVC

RVC aims at providing a system-level specification for existing and future MPEG standards [11]. In RVC, an abstract video decoder is built as a block diagram in which blocks define processing entities, indifferently called *actors* or Functional Units (FUs), and connections represent the data flow between actors. RVC provides both a normative standard library of FUs described with the RVC-CAL language, and a set of decoder descriptions expressed as networks of FUs using the FU Network Language (FNL).

RVC-CAL is a subset of the CAL Actor Language (CAL) described by the RVC standard. It restricts the datatypes, operators, and features that can be used when describing an FU. CAL [3] is an actor-oriented language created as part of the Ptolemy project [2], a Java framework that supports heterogeneous modeling, simulation, and design of concurrent dataflow systems made of actors.

FNL is an XML dialect with which networks can be described. FNL is based on a format used in the Open Dataflow environment[1] and standardized by MPEG. FNL supports local variable declarations, instantiation of blocks with parameters, and hierarchy: Blocks can be FUs or networks. Contrarily to other languages supported by OpenDF, FNL cannot create networks *programmatically* (e.g. with for loops).

2.1 RVC-CAL Language

An RVC-CAL actor is a modular component that encapsulates its own *state*. It can neither access nor modify the state of any other actor. An actor performs computation as a sequence of atomic steps called *firings*. During a firing, computation is done by one action, chosen among the several actions an actor may have. An action defines the amount of *tokens* consumed on the input, may change the actor state, and may output tokens using a function of inputs tokens and actor state.

Figure 1 shows an actor A that makes use of both functional and imperative features of RVC-CAL to extract red, green, blue components and luminance out of a pixel where colors are packed together. The actor has a parameter COMPUTE_Y that states whether luminance should be computed, and several constants that contains the masks and shift offsets for the different components. Actor A's single anonymous action requires *COUNT* tokens on the PIX input port to fire.

The body of an action is the same as a procedure's body in most imperative programming languages. Variables may be declared and initialized before any statement. Statements may be conditionals (if/then/else), loops (for/while), calls to functions and procedures, and assignments to local and state variables, either scalar or array. The language is strictly *structured*, in the sense that gotos are not allowed. Similarly it is not possible to break, to continue or to return a value. Needless to say, an actor cannot exit either.

RVC-CAL also borrows from functional programming. For instance *pure* (side-effect free) functions are distinct from imperative procedures. The expressions that can be written in conditions and in the right-

```
actor A (bool COMPUTE_Y) uint(size=24) PIX ⇒
  uint(size=8) R, uint(size=8) G, uint(size=8) B,
  uint(size=8) Y:

  int RSHIFT = 16; int RMASK = 255;
  int GSHIFT = 8; int GMASK = 255;
  int BSHIFT = 0; int BMASK = 255;
  int COUNT = 8;

  action: PIX:[pix] repeat COUNT ⇒
    R:[r] repeat COUNT,
    G:[g] repeat COUNT,
    B:[b] repeat COUNT,
    Y:[y] repeat COUNT
  var
    int i := 0
  do
    // imperative version to compute R, G, B
    while i < COUNT do
      r[i] := bitand(rshift(pix[i], RSHIFT), RMASK);
      g[i] := bitand(rshift(pix[i], GSHIFT), GMASK);
      b[i] := bitand(rshift(pix[i], BSHIFT), BMASK);

      i := i + 1;
    done

    // functional version to compute Y
    y :=
      if COMPUTE_Y then
        [rshift(
          66 * r[i] + 129 * g[i] + 25 * b[i] + 128,
          8) + 16 :
          for int i in Integers(1, COUNT) ]
      else
        [ 0 : for int i in Integers(1, COUNT) ]
      end;
  end

end
```

Figure 1 A sample actor showing both RVC-CAL programming styles.

[1] OpenDF is available on http://opendf.sf.net.

hand side of assignments resemble what is found in functional languages. They include `if/then/else` conditional expressions as well as *generators*. A generator is a kind of inline `for` loop that creates a list whose members are described by an expression.

The differences that are most significant between CAL and RVC-CAL are: (1) RVC-CAL supports only six types, which are divided into four primitive types (**bool**, **float**, **int**, **uint**) and two extended types (**List**, **String**) (2) all variables must be typed, this means that integers, signed or not, must have a size, and lists must be declared with the type of their elements and their maximum size (3) type parameters (called *generics* in Java or *templates* in C++) such as **T** in **actor A [T]** must not be used in type expressions such as **List[T]** (4) advanced features of CAL are prohibited, such as channel selectors and multi-ports, or lambda-functions.

For instance CAL allows developers to use λ-expressions. An example of a λ-expression is $f(x, y) = x + y$, where f can be evaluated partially: We can write $g = f(5)$, which is equivalent to declare $g(y) = 5 + y$. Compilers for languages that support λ-expressions must use *closures* made of the code of the expression and a set of variables and their values at the time the closure is created. Hardware synthesis of programs written in a functional language is easier if the language is restricted, especially if closures are forbidden [4]. Indeed, closures require dynamic memory allocation in the general case, something which is trivial in a software environment but requires a heavy machinery with hardware [15].

2.2 Semantics

In this document, we use the following conventions: *a* designates an action and the set of all actions inside an actor is written $\mathcal{A}$. In the actor `A` of Fig. 2, $\mathcal{A}$ equals to the set $\{a.x, a.y, b, c\}$.

Actions may be tagged. A tag is a non-empty list of identifiers separated by colons. The tag of an action a is written t_a. $|t_a|$ denotes the length of t_a. The empty tag ε verifies $|\varepsilon| = 0$. The set of non-empty tags of an actor is denoted T. There is a prefix relation, noted $\sqsubseteq$, between tags: $t \sqsubseteq t'$ means that t is a prefix of t'. For instance with tags `a` and `a.x` from actor `A` shown in Fig. 2, we have `a` $\sqsubseteq$ `a.x` and `a` $\sqsubseteq$ `a`. A set of actions that start with the same tag as an action a is described as follows:

$$\hat{t_a} = \{a_x \in \mathcal{A} | \, t_a \sqsubseteq t_{a_x}\} \tag{1}$$

```
actor A () uint(size=16) I1, uint(size=8) I2 ⇒
  int(size=32) O :

  // t and u are int(size=16)
  // v is List(type:int(size=8), size=5)
  a.x: action I1:[t, u], I2:[v] repeat 5 ⇒
    O:[ [t, u] + [0] + v ] repeat 8

  a.y: action ⇒ end

  b: action I1: [ i ] ⇒ end

  c: action ⇒ end

  priority
    b > a;
    a > c;
    a.x > a.y;
  end

  schedule fsm s0 :
    s0 ( b ) --> s0;
    s0 ( a ) --> s1;
    s1 ( c ) --> s0;
  end
end
```

Figure 2 A sample actor showing action-level control structures.

An action may have firing conditions, called *guards*, where the action firing depends on the values of input tokens or the current state. When an actor fires, an action has to be selected based on the number and values of tokens available and whether the guard is true. Action selection may be further constrained using a *Finite State Machine* (FSM), to select actions according to the current state, and *priority* inequalities, to impose a partial order among action tags. An FSM is defined by the triple (S, s_0, δ) where S is the set of states, $s_0 \in S$ is the initial state, and δ is the state-transition function: $\delta : S \times T \rightarrow S$. Note that a state transition allows a set of actions obtained with $\hat{t}$ from equation 1 to be fireable. Priorities have the form $t_1 > t_2$. These inequalities induce a binary relation on the actions as follows:

$$\begin{aligned} a_1 > a_2 \Leftrightarrow \; & \exists \, t_1, t_2 : t_1 > t_2 \;\wedge\; a_1 \in \hat{t_1} \;\wedge\; a_2 \in \hat{t_2} \\ & \vee \exists \, a_3 : a_1 > a_3 \;\wedge\; a_3 > a_2 \end{aligned} \tag{2}$$

FNL and RVC-CAL are expressive enough to create dataflow programs that follow a variety of computation models which differ by the trade-off they offer between expressive power and analyzability [2]. A network can be executed with the *Synchronous Dataflow* (SDF) model [9] if all actions inside an actor have the same token rates and at least one action may fire when actor is executed.

An important property of the SDF model is that liveness and boundedness can be decided at compile-time. Actors can be scheduled statically at compile-time and the memory requirement may be fixed a priori.

If actors contain state- and/or data-dependent firing conditions (called dynamic actors for short), then Dataflow Process Networks (DPN) [10] must be used. A DPN contains actors that communicate with each other using unidirectional FIFOs, where reads are blocking and writes are non-blocking. Contrarily to Kahn processes [7], actors may test an input for the absence of data. DPNs must be scheduled dynamically, hence actors are scheduled at runtime by an **actor scheduler**.

Each time an actor is scheduled, if the input tokens available and the actor state allow it, an action is fired. Testing an action's fireability is done by an **action scheduler** according to Algorithm 1, which is a reformulation of the conditions to fire an action as described in the CAL Language Report [3].

Algorithm 1 selects an action as follows. If the actor has a Finite State Machine (FSM), the set of **eligible** actions at runtime is found by Algorithm 2, otherwise the set equals $\mathcal{A}$. The subset of eligible actions for which the **is_activable**(a) function is `true` form the **activated** set. This function returns true if there are enough tokens to execute action a and the guard of action a evaluates to `true`. The set of **fireable** actions is computed by Algorithm 3 as the set of activated actions with a priority greater or equal to the priority of all other actions. Finally, the action selected to *fire* is any fireable action.

```
Input: $\mathcal{A}$, $\delta$, current state
Data: eligible, activated, fireable
Output: action selected
if actor has an FSM schedule then
    eligible ← select_eligible($\mathcal{A}$, $\delta$, current state);
else
    eligible ← $\mathcal{A}$;
end
activated ← $\{a \in$ eligible $\mid$ is_activable$(a)\}$;
fireable ← select_fireable(activated);
action selected ← any action in fireable;
```

Algorithm 1 Action Scheduler

Algorithm 2 founds the set of eligible actions. This set contains the actions that can be fired according to the FSM and the current state, in addition to all anonymous actions. This allows high priority anonymous actions to run outside an FSM whenever needed and can be used when one wishes to execute a given action from different states without changing state and without writing transitions.

```
Input: $\mathcal{A}$, $\delta$, current state
Data: transitions
Output: eligible actions
transitions ← $\{t \mid (s_f, t, s_t) \in \delta$ where $s_f =$ current state$\}$;
eligible ← $\{a \in \mathcal{A} \mid t_a = \varepsilon \vee a \in \hat{t}$ where $t \in$ transitions$\}$;
```

Algorithm 2 select_eligible

With proper data structures, the set of **transitions** is computed by **select_eligible** in $\mathcal{O}(1)$, and if the subsets $a \in \hat{t}$ have been pre-computed at compile-time the **eligible** actions are found in $\mathcal{O}(n)$ on the order of **transitions**.

Algorithm 3 determines the set of **fireable** actions at runtime in a time $\mathcal{O}(n^2)$ on the order of **activated**.

```
Input: activated, priorities
Output: fireable actions
fireable ← $\{a_1 \in$ activated $\mid \forall\, a_2 \in$ activated, $a_1 \geq a_2\}$;
```

Algorithm 3 select_fireable

2.3 Motivations and Issues for Software Code Generation

The main motivation for a software code generator for RVC-CAL was that there did not exist one even though it is necessary for successful development with the language. Cal2HDL [6] is able to generate high-speed VHDL from CAL, but this is far from an ideal solution for two reasons. On the one hand synthesis time is too long when developing an application. Simulation is an answer, but it runs very slowly, which increases development time. A software code generator is somewhere in-between by generating reasonably fast code in a reasonable amount of time. On the other hand the VHDL language is used as a description language for programmable logic such as FPGAs, and FPGAs are typically used as coprocessors to speed up repetitive computations. A software code generator is thus also necessary to obtain the software part of mixed hardware/software programs from RVC-CAL.

Translating actors and networks to software raises a number of issues. Contrarily to hardware code generation, having one thread per action in a software code is not efficient. FPGAs can execute several pieces of code in a truly parallel manner. Multi-core processors also provide true parallelism to a lesser extent, but it comes at a far higher cost because of the necessary synchronizations between cores. To lower the impact of this cost the pieces of code executed at the same time must be large enough, yet data dependencies limit the amount of available parallelism.

3 Cal2C Software Code Generator

Cal2C is open-source software under a BSD-like license programmed in Objective Caml, a functional ML-like language. Cal2C takes as input CAL actors and FNL networks and generates one C file per actor and a single C/C++ file from all the networks. We chose to use C as the target language for our software code generator because this language has become universal for software development and as a consequence there are libraries and compilers available for almost every platform. Figure 3 presents an overview of the process detailed in the following sections.

A hierarchical network composed of several networks and actors is flattened, and transformed to a C/C++ actor scheduler (Section 3.1). Each CAL actor is parsed to an abstract representation called Abstract Syntax Tree (AST), which is transformed by instantiation using parameters from the flat network, and translated to C (Section 3.2). The structure of the code generated is discussed in Section 3.3.

3.1 FNL Code Generation

Flattening an FNL network facilitates the actor code generation. Actors instances are renamed to ensure they have unique names, and actors are *closed*, which means their parameters are replaced by constant values. This has three consequences: (1) there can be a direct mapping between an actor instance and a C file generated, (2) the state variables of an actor can simply become global variables, (3) the generated code is more efficient because all uses of a parameter can be replaced by its value.

On the contrary, dealing with a hierarchical network is more complicated. Parameters have to be carried all the way through the hierarchy. The state of actors must be dynamically allocated each time an actor is instantiated, and given as a parameter to every action of the actor. The code that would be generated for a given network would be cleaner however, because there would be less code and it would be more structured. We deliberately chose to have a more efficient and cleaner code in actors at the expense of the network generated code's quality.

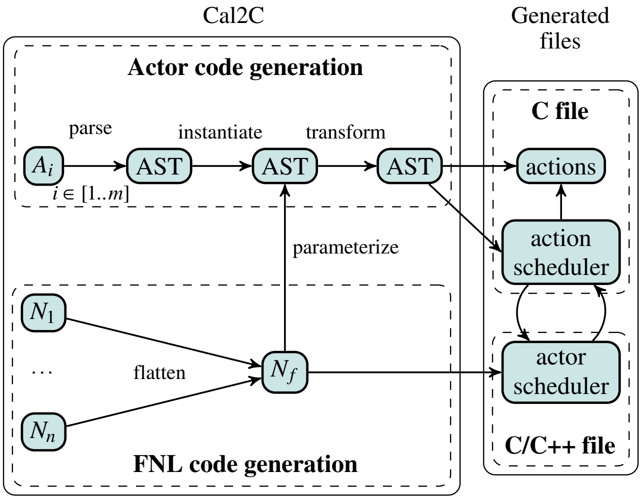


Figure 3 Code generation for *m* actors and *n* networks.

The flat network is used to automatically create an actor scheduler written in SystemC [5], a uniprocessor simulation framework. We chose SystemC because it provides us with a user-level scheduler, threads, FIFOs, and a model that closely matches ours: An actor becomes a `SC_MODULE`, input and output ports `sc_ports`, connections `sc_fifos`. A restriction of SystemC compared to FNL networks is that broadcast (multiple connections from a single output port to several input ports) is not supported, so special "broadcast" actors are inserted.

The *actor scheduler* implements Dataflow Process Networks (DPN) semantics on top of the SystemC model. `sc_fifos` cannot peek data, so it is necessary to add a buffer per FIFO to store the tokens peeked. Also, `sc_fifos` have blocking reads and writes while the DPN model has non-blocking writes. Therefore, after an action is found to be fireable, the action scheduler must test whether there is enough room to store the tokens the action will produce. Note that contrarily to RVC-CAL, unrestricted CAL does not make it possible to know in advance the number of tokens that may be produced by an action. Lack of room in output ports means that writing would block, which is not allowed by DPN semantics. The action scheduler thus simply indicates that the actor cannot fire and that it must `wait`. At this point, SystemC's scheduler suspends the actor, and attempts to fire another one picked from the list of currently sleeping actors.

SystemC `sc_fifos` admit a maximum size that must be specified at compile-time. FIFOs should be big enough for the network to be executed without artificially deadlocking, which occurs when an actor cannot write to a FIFO and no other actor can be fired to consume tokens from the same FIFO. With this in mind, choosing too big a size for every FIFO in the network will induce high latency and low average throughput with the current actor scheduler implementation. The user can use a default size for all FIFOs and fine-tune some particular FIFOs if necessary.

The *action scheduler* of each actor needs to check the presence and values of tokens on its input ports, get tokens from its input ports, and put tokens on its output ports. All these operations are dependent of the API that is used by the actor scheduler. To keep the actor code generic and actor scheduler-agnostic, actor scheduler and action scheduler exchange data through a set of well-defined functions. These functions are declared in the action scheduler and defined (implemented) by the actor scheduler. The functions have the following signatures:

- **int** `hasTokens_`A_i`_`p_j(**int** n) returns 1 if the input port p_j of actor A_i has at least n tokens present and 0 otherwise.
- **void** `peek_`A_i`_`p_j(**int** n, **T**[ ] tokens) peeks n tokens from the port p_j of actor A_i and puts them into *tokens*.
- **void** `read_`A_i`_`p_j(**int** n, **T**[ ] tokens) reads n tokens from the port p_j of actor A_i and puts them into *tokens*.
- **int** `hasRoom_`A_i`_`p_j(**int** n) returns 1 if the output port p_j of actor A_i has room for at least n tokens and 0 otherwise.
- **void** `write_`A_i`_`p_j(**int** n, **T**[ ] tokens) writes n *tokens* to the port p_j of actor A_i.

Note that the functions `hasTokens`, `peek_`A_i`_`p_j, and `read_`A_i`_`p_j must use a "snapshot" of the tokens available on all input ports of actor A_i at a time t because the firability of all actions shall be evaluated in the exact same conditions. This is the case with our implementation of DPN using SystemC because we allow the scheduler to switch to another actor only when the so-called actor cannot fire. This guarantees that the state of FIFOs will not change while an action scheduler examines them.

3.2 Actor Code Generation

The first stage of the compilation process is to parse each CAL actor to an Abstract Syntax Tree (AST). To this end we used a LL(k) parser bundled with OCaml called Camlp4. The grammar parsed is embedded in the code where a Syntax Directed Translation generates a node for each grammar rule or group of rules. For instance a simplified version of the AST generated for the CAL statement `x := if a > b then a else b end` is shown in Fig. 4.

After parsing the AST is modified when the actor is instantiated. Instantiation closes the actor i.e. it removes its parameters and replaces them by local variable declarations whose values are specified in the parent network.

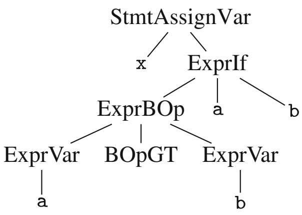


Figure 4 AST of `x := if a > b then a else b end`.

The next step is a series of transformations to the AST. The first one is type checking. It begins by annotating every expression in the AST with the type found according to the inference rules of the type system. An inference rule has the general form:

$$\frac{\Gamma \vdash P}{\Gamma \vdash expr : t} \tag{3}$$

It is read as follows: under the assumptions about the types of the variables in Γ, if the premises P are true, then the expression *expr* is well-typed and has type t. If an expression cannot be typed it is invalid according to the type system and it is not possible to generate code for the actor. If all expressions are well-typed, Cal2C checks that expressions assigned to variables have types that are compatible with the types of the so-called variables. If this is not the case, then compilation stops.

Once the AST is proven to be correctly typed, we apply a constant propagation algorithm. Such an algorithm is able to find values that are constant for all executions and propagate them through the program. Constant propagation is necessary to ensure a successful compilation of the C code. Variables of an actor can be initialized from the initial value of other variables of the actor, especially the variables initialized from parameters that are also variable declarations at this stage. Actor variables are translated to C global declarations, but in C global declarations cannot reference other global variables, even if they are constant. Propagating those constants solves the problem.

The last transformation converts the typed CAL AST to an Intermediate Representation closer to C. To our knowledge, CIL [12] (C Intermediate Language) is the framework that is best-suited for our purpose because it has a robust Intermediate Representation and the code it generates is a lot more readable than what SUIF [16] or LLVM [8] can generate. Before converting CAL to CIL names of variables, actions, etc. must be altered to be valid C identifiers. Some characters that are allowed in CAL identifiers but not in C (for instance '.' or '$') must be replaced. After that, Cal2C transforms CAL functional expressions into CIL imperative constructs: `if`s for if-then-else expressions, `for` loops for list generators. In the process temporary variables are created to hold temporary results in compound expressions. Figure 5 presents the CIL representation obtained from the CAL AST shown in Fig. 4. C code is obtained by calling the pretty-printer of the CIL API on the CIL tree.

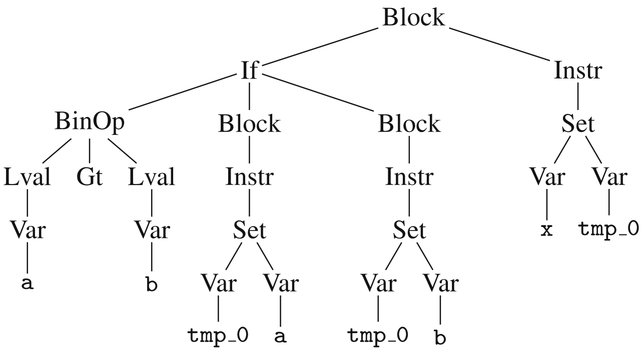


Figure 5 CIL AST obtained from the CAL AST of Fig. 4.

3.3 Structure of the Code Generated for an Actor

As shown in Fig. 3 the code generated for actions is separated from the code generated for the action scheduler. Each action becomes a functionally equivalent C function, and the action scheduler is implemented in yet another function. The rationale for separating actions from the action scheduler is two-folded: (1) the generated code is smaller than if the code for actions were generated *inside* the action scheduler, because actions may be referenced several times in a Finite State Machine (2) the generated code is easier to read because it has a structure similar to the original CAL code. Additionally, this does not impact performance because an optimizing compiler will automatically inline small-enough functions.

Figure 6 shows how the `a.x` action of actor `A` detailed in Fig. 2 is translated to C. The function wears a name deduced from the CAL name with respect to the rules defined in Section 3.2, i.e. the dot in `a.x` is replaced with an underscore. The function starts by reading tokens from the ports it uses, and puts them into local arrays. The function then executes the C equivalent of the CAL action body and finally writes tokens to output ports.

The action scheduler for an actor *A* is a function *A_scheduler*() called every time an actor fires. This function is an implementation of Algorithm 1 described in Section 2.2 that selects a *fireable* action if input tokens and actor state allow it. Because action scheduler functions are virtually called all the time during the execution of a network, it is crucial they be as fast as possible.

Figure 6 Action a.x translated to a functionally-equivalent C function.

```
void a_x()
{
  unsigned short I1[2];
  unsigned char I2[5];
  int O[8];

  read_A_I1 (2, I1);
  read_A_I2 (5, I2);

  ...

  write_A_O (8, I2);
}
```

The following describes how to generate optimal code to compute the different steps of Algorithm 1. The implementation of **select_eligible**(A) is detailed in Section 3.5. This function selects the actions eligible according to the current FSM, or every action if the actor has no FSM. **is_activable**(a) is translated to C conditions that are evaluated at runtime each time the action scheduler is run, and are true if (1) there are enough tokens to execute action *a* and (2) the guard of action *a* evaluates to `true`. Testing the number and values of tokens on port j of actor i is done by calling the functions from the actor scheduler presented in Section 3.1, `hasTokens_`A_i`_`p_j and `peek_`A_i`_`p_j. The last step is finding the **fireable** actions, i.e. the actions in **activated** with the highest priority. This is done by the **select_fireable** function described in Section 3.4.

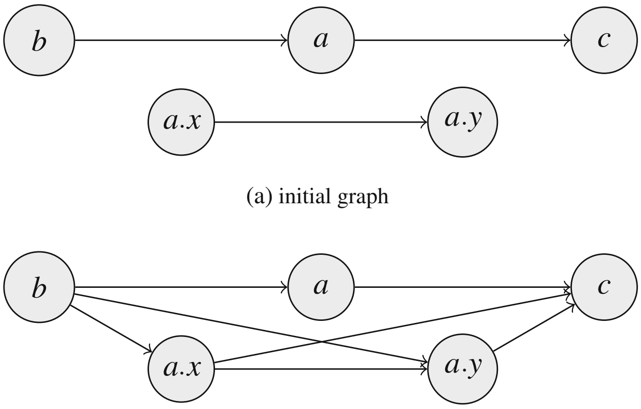
3.4 Computing **Select_Fireable** at Compile-Time

Computing the set of *fireable* actions can be done in the generated code with if-then-else statements by sorting actions by priority at compile time. To this end a total order of priorities is obtained from the partial order priority relations by topologically sorting a Directed Acyclic Graph (DAG) created by Algorithm 4.

```
Input: priorities
Output: $G = (V,E)$ as a DAG
$V \leftarrow \emptyset$;
$E \leftarrow \emptyset$;
// creation of the initial graph
for each $t_1 > t_2$ relation do
    $V \leftarrow V \cup \{t_1\} \cup \{t_2\}$;
    $E \leftarrow E \cup \{(t_1,t_2)\}$;
end
// transformation into the final graph
for $t_{a'} \in V$ do
    if $\exists\, t_a \sqsubset t_{a'} \wedge t_a \in V \wedge max(|t_a|)$ then
        $E \leftarrow E \cup \{(t_p,t_{a'}) \mid (t_p,t_a) \in E\}$;
        $E \leftarrow E \cup \{(t_{a'},t_s) \mid (t_a,t_s) \in E\}$;
    end
end
```

Algorithm 4 Creating a DAG from priorities

Algorithm 4 works on a directed graph $G = (V, E)$ where $V \subseteq T$ is the set of vertices and $E \subseteq V^2$ is the set of edges. A vertex is a tag involved in a priority relation. An edge from *source* to *target* is equivalent to the priority relation *source* > *target*. The graph G is initially filled from the priority relations. The second



(a) initial graph

(b) final graph

Figure 7 From a sparsely-connected priority graph to a DAG (**a**, **b**).

`for` loop iterates over each tag $t_{a'}$ present in the graph. If there exists in the graph a tag t_a that is the longest strict prefix of $t_{a'}$, the algorithm adds edges between the predecessors of t_a and the tag $t_{a'}$ and between the tag $t_{a'}$ and the successors of t_a.

Figure 7 presents the initial graph created from the priorities of the sample actor given in Fig. 2 after the first step of Algorithm 4 and the final graph after the second step of the algorithm. The final graph is sorted by topological order that gives a total order from the priorities that were stated: $[b\,; a.x; a.y; c]$. Since the actor does not define any relations between a and $a.x$ or a and $a.y$, the topological order of the graph will yield either $[b\,; a; a.x; a.y; c]$ or $[b\,; a.x; a; a.y; c]$. In the present example this is not relevant because there is no action a. If there was one however, this would lead to non-deterministic behavior, in which case the code generator or simulator is free to choose whether to test a before $a.x$. In our case the choice is made by the underlying graph library **ocamlgraph** [1], using a lexical order over vertices with the same topological rank.

3.5 Computing **Select_Eligible** at Compile-Time

By transforming **select_eligible** into adequate C code, the set of eligible actions is computed in $\mathcal{O}(1)$.

In the translation of an FSM (S, s_0, δ), the set of states S becomes a C `enum` filled with the states. State names are modified to provide valid C identifiers. Cal2C creates a `switch` statements with as many `cases` as there are states; they are labeled with created identifiers. The current state is available through an integer variable named `state` whose range is the `enum.state` is initialized to the initial state s_0.

```
switch (state) {
  case s0:
    if (is_activable_A_b()) {
      A_b();
      state = s0;
    } else if (is_activable_A_a_x()) {
      A_a_x();
      state = s1;
    } else if (is_activable_A_a_y()) {
      A_a_y();
      state = s1;
    }
    break;

  case s1:
    if (is_activable_A_c()) {
      A_c();
      state = s0;
    }
}
```

Figure 8 FSM of Fig. 2 translated to C.

Each triple (s_f, t_{a_i}, s_t) of the state transition function δ is transformed as follows. Inside the `case` s_f, the set of eligible actions is defined as

$$eligible = \bigcup_{i=1}^{n} \hat{t}_{a_i} \tag{4}$$

As explained in Section 3.4, actions of $eligible$ are sorted by priority and translated to if-then-else statements. The state-transition is done by an assignment `state` $= s_t$; added at the end of each `then` branch. As an example, the translation of the FSM of the actor of Fig. 2 is presented on Fig. 8.

4 Case Study: Two MPEG-4 SP Decoders

Cal2C is able to successfully translate two different MPEG-4 Simple Profile decoder descriptions. These decoders differ both from the topology of the network and from the granularity of actors. The first decoder D_1 contains 54 actors that come from the Video Tool Library, the normative FU database [14]. Y, Cb and Cr components are decoded separately, exposing coarse grain parallelism in the network. Similarly actors expose only coarse-grain parallelism. For instance, the Inverse Discrete Cosine Transform (IDCT) is implemented with a single actor that performs the IDCT using an 8-point based sequential algorithm in a single action.

The second decoder D_2 is a proprietary decoder from Xilinx, available in the OpenDF project [6]. Being targeted at hardware it emphasizes fine-grain parallelism and pipelining. It decodes Y, Cb and Cr components serially, and its IDCT is a network that performs computations at pixel level. This decoder totals up 32 actors.

Table 1 Runtime of MPEG-4 SP decoder's descriptions.

Decoder	Throughput		
	OpenDF	Cal2C	Cal2HDL
D_1	20	4318	–
D_2	16	3614	290000

The video sequence used in the experiment is "Foreman" with 300 frames, 176 × 144 (QCIF), IPPP mode, encoded at 30 fps. These experiments were run on an Intel E8500 Core2Duo processor at 3.2 GHz. Only one of the cores was used by generated software because of SystemC. The throughputs of the two descriptions are expressed in macroblocks per second and compared in Table 1. The first column gives the throughput of the CAL model simulated using the Open Dataflow runtime environment. The second column gives the throughput of the binary programs compiled from the code generated by Cal2C using Microsoft Visual Studio 2008 with maximal optimizations. The third column gives the throughput of the circuit synthesized from Cal2HDL on a Xilinx Virtex-2 (this result comes from [6]).

The fidelity of the generated code can be tested in two ways. The first one is a general method that allows a single actor to be tested. In the simulator, tokens read and written by an actor A to test are recorded in separate files, say f_{in} and f_{out}. The designer then generates code for a simple network that contains three actors, a source actor that produces tokens read from f_{in}, the A actor, and a sink actor that compares tokens produced by A against tokens in f_{out}. The second way of testing the fidelity of the generated code for a whole decoder is to match decoded macroblocks against a reference YUV file.

Results show that not only is Cal2C robust enough to successfully generate code for both decoders, but it also generates code that decodes the sample QCIF sequence at 30 fps in both cases. Results also indicate that the code generated from coarse grain actors performs better than code generated from fine grain actors. This is due to the fact that in our current implementation an actor performs computation in its own thread, which means the coarser the actor the more work it will do without triggering a context switch.

5 Perspectives

As said in Section 3.1, SystemC is a good choice for easily implementing DPNs, but it is far from an ideal solution. Each actor, or `SC_MODULE`, is executed in its own thread. Each time an actor must `wait`, several context switches occur because the SystemC scheduler does not take the dataflow nature of the model into account, and schedules any sleeping actor even if it could clearly not fire (e.g. no data on its input ports). When networks contain tens of threads (like the decoders we tested), those context switches adds a visible overhead. Another limitation of SystemC is that FIFOs can be read from/written to only one token at a time by calling read/write functions, which is a double source of overhead. The cost of calling a function only to load/store an integer is not negligible, and neither are the memory copies between SystemC FIFOs and peekable buffers.

We believe that it is necessary to implement our own user-level scheduler. A decent DPN scheduler should have the three following characteristics: (1) no use of threads, (2) specific to the network it is generated for, (3) direct access to peekable FIFOs via pointers.

The DPN model was chosen to be as general as possible. Indeed, our initial goal was to create a compiler that translated any actor that could be written with the RVC-CAL language. The selected computation model considers all actors as dynamic and schedules them at runtime. This overhead may be reduced by using restricted dataflow computation models when possible, such as the SDF model. Even though a network may not be statically schedulable as a whole, regions of the network may be. The SDF model allows for compile-time analysis, making it possible to statically schedule such regions at compile-time with bounded memory. The scheduler is turned into a simple sequencer that executes periodically actors in a predefined order inside a single thread. We plan to implement analysis techniques in Cal2C that would detect such statically schedulable regions, which may improve significantly the efficiency of the compiled program.

As a matter of fact, the SDF model also has advantages for scheduling and code generation in the context of multi-core targets. For instance, PREESM [13] is a tool that schedules and maps actors at compile-time over multicore targets and generate optimized distributed code (including communication and synchronization between cores). Our final goal is to provide a complete framework using Cal2C, PREESM and Cal2HDL to generate efficient code for heterogeneous platforms with multicores and programmable logic devices.

6 Conclusion

This paper presents a software code generator that automatically translates dataflow programs written in RVC-CAL to C and a SystemC model. We show how to generate readable C code from the actions and action scheduler of actors, and define a clear interface between the actor scheduler and the action schedulers. Cal2C successfully translates two descriptions of an MPEG-4 SP decoder. The results obtained so far show the efficiency of the generated code considering it uses a uniprocessor simulation framework. Perspectives are given to improve the efficiency of the generated code.

Another contribution is that the results presented here show that a network of RVC-CAL FUs provided by the MPEG group can be translated automatically into an efficient software implementation. This specification can also be translated into RTL code targeting FPGAs leading to smaller and faster design than a handmade VHDL reference design [6]. In our opinion, RVC-CAL is a high-level language well suited for functional description and for fast prototyping methods that aim to provide hardware/software optimized implementations.

References

1. Conchon, S., Filliâtre, J. C., & Signoles, J. (2008). Designing a generic graph library using ML functors. In M. Morazán (Ed.), *Trends in functional programming* (Vol. 8). Oxford: Intellect
2. Eker, J., Janneck, J., Lee, E., Liu, J., Liu, X., Ludvig, J., et al. (2003). Taming heterogeneity—the Ptolemy approach. In *Proceedings of the IEEE* (Vol. 91).
3. Eker, J., & Janneck, J. (2003). CAL language report. Tech. Rep. ERL Technical Memo UCB/ERL M03/48, University of California at Berkeley.
4. Frankau, S., & Mycroft, A. (2003). Stream processing hardware from functional language specifications. In *Proceedings of 36th Hawaii intl. conf. on systems sciences* (HICSS-36).
5. IEEE (2006). IEEE std 1666-2005 IEEE standard SystemC language reference manual. IEEE Std 1666-2005.
6. Janneck, J. W., Miller, I. D., Parlour, D. B., Roquier, G., Wipliez, M., & Raulet, M. (2008). Synthesizing hardware from dataflow programs: An MPEG-4 simple profile decoder case study. In *IEEE workshop on signal processing systems (SiPS 2008)* (pp. 287–292).
7. Kahn, G. (1974). The semantics of a simple language for parallel programming. In *Proceedings of IFIP'74* (pp. 471–475).
8. Lattner, C., & Adve, V. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 international symposium on code generation and optimization (CGO'04)*. Palo Alto, California.
9. Lee, E. A., & Messerschmitt, D. G. (1987). Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers, 36*(1), 24–35.
10. Lee, E. A., & Parks, T. M. (1995). Dataflow process networks. *Proceedings of the IEEE, 83*(5), 773–801.
11. Lucarz, C., Mattavelli, M., Thomas-Kerr, J., & Janneck, J. (2007). Reconfigurable Media Coding: A new specification model for multimedia coders. In *Proceedings of SIPS'07*.
12. Necula, G. C., McPeak, S., Rahul, S. P., & Weimer, W. (2002). CIL: An infrastructure for C program analysis and transformation. In *Proceedings of CC'02* (pp. 213–228).
13. Pelcat, M., Menuet, P., Nezan, J. F., & Aridhi, S. (2009). Scalable compile-time scheduler for multi-core architectures. In *Proceedings of design automation and test in Europe* (DATE'09).
14. Roquier, G., Wipliez, M., Raulet, M., Janneck, J. W., Miller, I. D., & Parlour, D. B. (2008). Automatic software synthesis of dataflow program: An MPEG-4 simple profile decoder case study. In *IEEE workshop on signal processing systems (SiPS 2008)* (pp. 281–286). Washington, D.C., U.S.A.
15. Séméria, L., Sato, K., & Micheli, G. D. (2000). Resolution of dynamic memory allocation and pointers for the behavioral synthesis from C. In *Proceedings of design automation and test in Europe (DATE'00)* (pp. 312–319). New York: ACM.
16. Wilson, R. P., French, R. S., Wilson, C. S., Amarasinghe, S. P., Anderson, J., & Tjiang, S. W. K., et al. (1994). Suif: An infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Notices, 29*(12), 31–37.

Matthieu Wipliez is currently a Ph.D. student at the Institute of Electronics and Telecommunications of Rennes (IETR) in France. In 2006, he graduated from the National Institute of Applied Sciences (INSA) of Rennes with a M.Sc. in computer science. His main research interests include dataflow programming, compilers, multiprocessing, and rapid prototyping.

Ghislain Roquier is currently a post-doc researcher in the GR-LSM Multimedia group at Ecole Polytechnique Fédérale de Lausanne in Switzerland. In 2005, he graduated from Université de Rennes I, France, with a M.Sc. in signal processing. He received the Ph.D. degree in electronics in 2008 from the Institut National des Sciences Appliquées de Rennes, France. His main research interests include design and implementation of heterogeneous real-time embedded systems, rapid prototyping methodologies, multimedia DSP.

Jean-François Nezan is an Assistant Professor at National Institute of Applied Sciences of Rennes (INSA) and a member of the IETR laboratory in Rennes. He received his postgraduate certificate in signal, telecommunications, images, and radar sciences from Rennes University in 1999, and his engineering degree in electronic and computer engineering from INSA-Rennes Scientific and Technical University in 1999. He received his Ph.D. degree in electronics in 2002 from the INSA. His main research interests include image compression algorithms, multiprocessor rapid prototyping and dataflow programming.