IMPLEMENTING STREAM-PROCESSING APPLICATIONS ON FPGAS: A DSL-BASED APPROACH

J. Sérot, F. Berry, S. Ahmed

UMR 6602 CNRS / Université Blaise Pascal
Clermont-Ferrand, France
email: {jocelyn.serot,francois.berry,sameer.ahmed}@lasmea.univ-bpclermont.fr

ABSTRACT

We introduce CAPH, a new domain-specific language (DSL) suited to the implementation of stream-processing applications on field programmable gate arrays (FPGA). CAPH relies upon the actor/dataflow model of computation. Applications are described as networks of purely dataflow actors exchanging tokens through unidirectional channels. The behavior of each actor is defined as a set of transition rules using pattern matching. The CAPH suite of tools currently comprises a reference interpreter and a compiler producing both SystemC and synthetizable VHDL code. We describe the implementation, with a preliminary version of the compiler, of a simple real-time motion detection application on a FPGA-based smart camera platform.

1. INTRODUCTION

Stream-processing applications – i.e. applications operating on the fly on continuous streams of data – generally require a high computing power. This is specially true of real-time image processing for instance, in which this computing power is in the range of billions of operations per second, often still beyond the capacity of general purpose processors (GPPs). Most of the computationally demanding tasks in these applications exhibit parallelism, making them good candidates for implementation on reconfigurable logic such as FPGAs.

But, in the current state-of-the art, programming FPGAs essentially remains an hardware-oriented activity, relying on dedicated Hardware Description Languages (such as VHDL or Verilog). Using these languages requires expertise in digital design and this practically limits the applicability of FPGA-based solutions.

As a response, a lot of work has been devoted in the past decade to the design and development of high-level languages and tools, aiming at allowing FPGAs to be used by programmers who are not experts in digital design. Fueled by new behavioral synthesis techniques and ever-increasing FPGA capacities, significant advances have been made in this area. But there’s still a gap between what can be described with a general purpose, Turing-complete, language and what can be efficiently and automatically implemented on an FPGA. In this context, we believe that a domain-specific language (DSL) can provide a pragmatic solution to this problem.

In this paper, we introduce such a DSL, called CAPH. By adopting a specific (purely dataflow) model of computation, CAPH aims at reducing the gap between the programming model (as viewed by the programmer) and the execution model (as implemented on the target hardware) and hence obtaining an acceptable trade-off between abstraction and efficiency requirements.

The remainder of the paper is organized as follows: In Section 2, we recall the issues raised by FPGA programming, make a brief survey of some existing solutions and explain why they are not satisfactory. Section 3 presents the general dataflow/actor oriented model of computation as a possible solution to the aforementioned issues. Section 4 introduces the CAPH language as a specific incarnation of the general dataflow approach. Section 5 is an overview of the suite of tools supporting the CAPH language. Section 6 gives a short and basic account on how the compiler works in order to generate efficient VHDL code. Preliminary experimental results are presented in Section 7. Related work is described and discussed in Section 8. Finally, conclusions are drawn in Section 9.

2. HIGH LEVEL LANGUAGES FOR FPGA

Hardware Description Languages (HDLs), such as VHDL or Verilog, are still widely used for programming FPGAs because they provide flexible and powerful ways to generate efficient logic. But these languages were designed specifically for hardware designers, which makes them unfamiliar for programmers outside this field. To circumvent this problem, a number of tools have been proposed - both from the

\[\text{CAPH is a recursive acronym for Caph just Aint Plain Hdl. It is also the name of the second brightest star in the constellation of Cassiopeia.}\]
Many of these tools propose a direct conversion of C code into (V)HDL. These include Handle-C [1], Stream-C [2], SA-C [3], SPARK [4] and Impulse-C [5]. These approaches suffer from several drawbacks. First, C programs sometimes (often) rely on features which are difficult, if not impossible, to implement in hardware (dynamic memory allocation for instance). This means that code frequently has to be rewritten to be accepted by the compilers. Practically, this rewriting cannot be carried out without understanding why certain constructs have to be avoided and how to replace them by "hardware-compatible" equivalents. So a minimum knowledge of hardware design principles is actually required. Second, C is intrinsically sequential whereas hardware is truly parallel. So the compiler has to first identify parallelism in the sequential code and then map it onto the target hardware. In the current state-of-the-art, this cannot be done in a fully automatic way and the programmer is required to put annotations (pragmas) in the code to help the compiler, which adds to the burden. Finally, the code generally has to undergo various optimizations and transformations before the actual HDL generation. These optimizations and transformations vary from high level parallelization techniques to low level scheduling. The low level optimizations can be beneficial to any algorithm, but the high level optimizations are specifically suggested in the context of one field and would not give performance gains in other domains [6]. Moreover, with most of the existing tools (Handle-C, Impulse-C, Catapult-C), transformations and optimizations require inputs from the programmer [7], who therefore must have a rather good knowledge in digital design.

3. DATAFLOW/ACTOR ORIENTED PARADIGM

The solution to the problems raised by C-like approaches to FPGA programming requires a shift in programming paradigm. In particular, it seems crucial to reduce the gap between the programming model (as viewed by the programmer) and the execution model (as implemented on the target hardware). The dataflow/actor paradigm offers a way to achieve this goal. This section recalls the key features of this paradigm and why it is naturally suitable for FPGA or reconfigurable devices.

The common and basic underlying concept is that applications are described as a collection of computing units (often called actors) exchanging tokens through unidirectional channels (typically FIFOs). Execution occurs as tokens flow through channels, into and out of actors, according to a set of firing rules. In the classical, strict, dataflow model, the firing rules specify that an actor becomes active whenever tokens are available on all of its input channels and token(s) can be written on its output channel(s). When this occurs, input tokens are consumed, result(s) are computed and produced on the output channel(s). This strict firing model has been latter extended to accommodate more complex and sophisticated scheduling strategies (see Section 8).

The advantages of the dataflow model are well-known. First, it basically relies on a representation of applications in the form of dataflow graphs (DFGs), which is intuitive, well-understood by programmers (esp. in the field of digital signal processing), and amenable to graphical representation and manipulation. Second, the behavior of each actor is defined independently of what occurs in the other ones, all parallel operations may be executed concurrently, without the risk of side-effects, hence allowing full exploitation of intrinsic data and control-level parallelism. Third, since several tokens are allowed to reside simultaneously on a channel (by implementing it using a FIFO typically), execution pipelining may be increased, each actor being able to produce one output token before the previous one has actually been consumed.

4. CAPH LANGUAGE

CAPH is based upon the dataflow/actor programming and execution model described in the previous section. An application to be implemented on a reconfigurable device is described as a network of computational units called actors, communicating with each other through FIFO channels, as depicted in Fig. 1, where the four actors are denoted A1, ..., A4. Interaction between actors is strictly limited to token exchange through channels, so that the behavior of each actor can be completely described in terms of actions performed on its inputs to produce outputs (no side effect, strictly local control).

As most of dataflow or actor-based models, CAPH actually relies on two levels of description: one to specify the behavior of each actor and the other to describe how actors are interconnected (the network topology). The originality of CAPH, compared to existing similarly based systems—such as those described in Section 8 in particular—lies in the languages supporting these two levels.
4.1. Describing actors

The description of the actor comprises its name, a possible list of (typed) parameters, a list of (typed) inputs and outputs, a list of local variables and a description of its behavior. The behavior itself is specified using a set of transition rules.

Each rule consists of a pattern, involving inputs and/or local variables and a corresponding expression, describing modifications of outputs and/or local variables. The choice of the rule to be fired is done by pattern-matching. Consider, for example, the actor `switch` described in Fig. 2. This actor alternately copies its first and second input to its output, as depicted on the right. Its behavior description in CAPH is given on the left. It has no parameter (()). Its behavior description in C

alternately copies its first and second input to its output, as described in Fig. 2. This actor should be read, precisely, as: if a value is available on input `a` and set `s` to `right` (resp. `left`), then read the same token on output `o`. The first (resp. second) rule says: If `s` is `left` (resp. `right`) and a value `v` is available on input `i2` (resp. `i1`) then read this value, write it to output `o` and set `s` to `right` (resp. `left`). The `'|'` symbol used in the pattern means that the corresponding input is not used.

4.2. Data representation

A key property for a programming model is its ability to represent arbitrarily structured data. For stream-processing applications, this structuring can be achieved by dividing the tokens, circulating on channels and manipulated by actors, into two broad categories: data tokens (carrying actual values) and control tokens (acting as structuring delimiters).

Fig. 2. An example of actor description in CAPH

![Fig. 2](image-url)

<table>
<thead>
<tr>
<th>actor switch ()</th>
</tr>
</thead>
<tbody>
<tr>
<td>in (i1:int)</td>
</tr>
<tr>
<td>out (o:int)</td>
</tr>
<tr>
<td>var s: (left, right) = left</td>
</tr>
<tr>
<td>rules</td>
</tr>
<tr>
<td>(left, v,</td>
</tr>
<tr>
<td>(right, _, v) -&gt; (v, left)</td>
</tr>
</tbody>
</table>

Fig. 3. The structured stream representation of a 4x4 image

![Fig. 3](image-url)

The structured stream representation of data nicely fits the stream-processing programming and execution models. Since the structure of the data is explicitly contained in the token stream no global control and/or synchronization is needed; this has strong and positive consequences both at the programming level (it justifies a posteriori the style of description we introduced in the previous subsection for actors) and the execution level (it will greatly ease the production of HDL code). Moreover, it naturally supports a pipelined execution scheme: processing of a line by an actor, for example, can begin as soon as the first pixel is read without having to wait for the entire structure to be received; this feature, which effectively allows concurrent circulation of successive “waves” of tokens through the network of actors is of course crucial for on-the-fly processing (like in real-time image processing).

Fig. 4 gives the code of an actor performing binarization of a structured stream of pixels (of an image, for instance). The threshold value is passed as parameter `t`. Pattern-matching is used to discriminate between control and data tokens. The rules can be read as follows: if input is a control token (`<` or `>`) then write the same token on output; if input is a data token, then write 0 or 1 on output depending on whether the associated value is greater than the threshold parameter or not.

---

2 The quote (' ') symbol is used to distinguish structured values (control or data) from unstructured (raw) values (like in the previous `switch` example). This distinction is reflected in the type of the input and output: `inc dc`, where `dc` is the type constructor for structured values. Hence, the last rule of the `thr: actor` should be read, precisely, as: if a value is available on input `a` and this value is a data token carrying value `v`, then produce a data token on output carrying value 0 or 1 depending on whether `v > t` or not.

---

3 Pop the value from the connected FIFO.

4 The `'|'` symbol can also be used in the right-hand side of a rule; it then means that no value is produced on the corresponding output.
actor thr (t:int)
in (a:int dc)
out (c:int dc)
rules (a) -> (c)
| '<' -> '<'
| '>' -> '>
| 'v' -> if v > t then '1' else '0

Fig. 4. An actor performing binarization on structured streams

4.3. Describing networks

In CAPH, the structure of the actors network is expressed using a small, purely functional language, for describing data flow graphs called FGN (Functional Graph Notation). The syntax and semantics of this language have been described in detail in [8] so this section is a minimal description of its possibilities.

The basic idea is that the network of actors is actually a dataflow graph (DFG) and that a DFG can be described by means of purely functional expressions. For example, the network depicted in Fig. 1 – in which A1, …, A4 are actors (viewed as functions from streams to streams), i an input stream and o an output stream – can be described with the following equations:

\[
\text{ndef} (x, y) = A1 \text{ i}
\]
\[
\text{ndef} o = A4 (A2 x, A3 y)
\]

where \( f \ x \) denotes application of function \( f \) to argument \( x \), \( (x, y) \) denotes a pair of values and the \text{ndef} keyword serves to introduce bindings.

Compared to other textual or graphical network languages, this notation offers a significantly higher level of abstraction. In particular it saves the programmer from having to explicitly describe the wiring of channels between actors, a tedious and error-prone task. Moreover, ill-formed networks and inconsistent use of actors can be readily detected using a classical Hindley-Milner polymorphic type-checking phase. Another advantage of “encoding” data-flow networks in a functional language – not demonstrated here, see [8] – is the ability to define reusable, polymorphic graph patterns in the form of higher-order functions, which offers an easy and safe compositional approach for building larger applications from smaller ones.

As a result, a CAPH program will comprise at least three sections (see section 7 for a complete example) : one section containing the definition of the actors, another one describing the network description and a last defining the input and output streams. An optional section (not discussed further here) is available to define global constants or functions.

5. THE CAPH TOOLSET

The current tool chain supporting the CAPH language is sketched on Fig. 5. It comprises a graph visualizer, a reference interpreter and compiler producing both SystemC and synthesizable VHDL code\(^5\).

The graph visualizer produces representations of the actor network in the .dot format for visualisation with the GRAPHVIZ suite of tools [9]. An example is given Fig. 9.

The reference interpreter is based on the fully formalized semantics of the language, written in axiomatic style. Its role is to provide reference results to check the correctness of the generated SystemC and VHDL code. It can also be used to test and debug programs, during the first steps of application development (in this case, input/output streams are read from/written to files). Several tracing and monitoring facilities are provided. For example, it is possible to compute statistics on channel occupation or on rule activation.

The compiler is the core of the system. It relies on an elaboration phase, turning the AST into a target-independent intermediate representation, and a set of dedicated back-ends. The intermediate representation (IR) is basically a process network in which each process is represented as a generalized finite-state machine (GFSM) and channels as unbounded FIFOs. Two back-ends are provided : the first produces cycle-accurate SystemC code for simulation and profiling, the second VHDL code for hardware synthesis. Execution of the SystemC code provides informations which

\[^5\text{Synthesis of the generated VHDL code is carried using third-party tools; we currently use the ALTERA Quartus II environment.}\]
are used to refine the VHDL implementation (for example: the actual size of the FIFOs used to implement channels).

The graph visualizer, the reference interpreter and the compiler all operate on an abstract syntax tree (AST) produced by the front-end after parsing and type-checking.

6. ELABORATION

Due to space limitations, we only give an overview of the elaboration process, focusing on the generation of VHDL code (generation of SystemC code follows the same principles). A detailed account is given in [8].

Elaboration involves two steps: first generating a structural representation of the actor network and then generating a behavioral description of each actor involved in the network.

Generating the structural representation of the actor network involves instantiating each actor – viewed as a black box at this level – and “wiring” the resulting instances according to the dependencies expressed by the functional definitions. The CAPH compiler uses a technique known as abstract interpretation to perform this. The basic idea is that the definitions of the program are “evaluated” and the each evaluation of a function bound to an actor creates an instance of this actor in the graph. Each wire is then instantiated as a FIFO channel.

Generating the behavioral description of each actor essentially consists in turning the set of pattern-matching rules into a finite state machine with operations (FSMD level). A full and formal account of this process is beyond the scope of this paper. We will illustrate it with a simple actor computing the sum of a list of values. Given the input stream \(< 1 2 3 >\), for example, this actor will produce the value 6. Fig. 6-a gives a possible CAPH description of this actor. Fig. 6-b is a FSMD transcription, in which the local variable st is used as a state value and the transitions are labeled with conditions/actions patterns. For example, the second rule of the actor says: if variable st equals S1, if a data value v is available on input a then do not output anything on c, add v to s and keep st to S1.

It is easy to translate this FSMD representation of an actor to VHDL (or SystemC) code. A clock is added (because we are targeting synchronous designs). Read (resp. write) operations on inputs (resp. outputs) are converted into signals controlling the FIFOs connected to these inputs/outputs. There are three signals for each input: dataout, empty and rd and three signals for each output: datain, full and wr. The signals empty (resp. full) tells whether the FIFO is empty (resp. full), i.e. ready for reading (resp. writing). Finally some states are inserted to generate the rd (resp. wr) signals which trigger the read (resp. write) operation, i.e. actually pop (resp. pushes) the data on the FIFO.

actor suml ()
in (a: int dc)
out (c: int)
var st: S0,S1=S0
var s : int
rules st,a,s-> st,c,s
S0,'<,_ -> S1,_,'_ 0
| S1,'v,s -> S1,_,'_ ,s+v
| S1,'>,s -> S0,s,_'

Fig. 6. Actor elaboration

The resulting VHDL code follows:

entity sum_act is
  port (a_empty: in std_logic;
a: in std_logic_vector(9 downto 0);
a_rd: out std_logic;
c_full: in std_logic;
c: out std_logic_vector(15 downto 0);
c_wr: out std_logic;
clock: in std_logic;
reset: in std_logic);
end sum_act;

architecture FSM of sum_act is
  type t_state is (S0,S01,S1,S11,S12);
begin
  process(clock, reset)
    variable s : std_logic_vector(15 downto 0);
    variable st : t_state;
    variable v : std_logic_vector(7 downto 0);
  begin
    if (reset='0') then
      st := S0; a_rd <= '0'; c_wr <= '0';
    elsif rising_edge(clock) then
      case state is
      when S0 =>
        if a_empty='0' and is_sos(a) then
          a_rd <= '1';
st := S01;
s := conv_std_logic_vector(0,15);
        end if;
      when S01 =>
        a_rd <= '0'; state <= S1;
      when S1 =>
        if a_empty='0' and is_data(a) then
          a_rd <= '1'; v := data_from(a);
s := s+v; st := S11;
        end if;
      when S11 =>
        if a_empty='0' and is_eos(a) then
          a_rd <= '1'; c := s;
c_wr <= '1'; st := S12;
        end if;
      end case;
  end process;
end sum_act;
when S11 =>
    a_rd <= '0'; st := S1;
when S12 =>
    a_rd <= '0'; c_wr <= '0'; st := S0;
end case;
end if;
end process;
end FSM;

The functions is_sos, is_eos, is_data and data_from are part of a package providing operations on structured values. They respectively tell whether their argument is a control value – ‘<’ (Start of Structure) or ‘>’ (End of Structure) – or a data value and, in the latter case, extract this value. Physically, the distinction between control and data values is encoded with two extra bits.

This example shows how the embedding of control tokens in the data streams simplifies scheduling by allowing it to be purely local.

7. EXPERIMENTAL RESULTS

We have experimented with a prototype version of the compiler using a very simple application as a test bench. The goal is to validate the overall methodology before moving to more complex algorithms and to identify key issues. The application is a simple motion detector operating in real-time on a digital video stream. Moving objects are detected by spatio-temporal changes in the grey-level representation of the successive frames and a a rectangular window is drawn around them.

The algorithm involves 1) computing the difference image between two consecutive frames 2) thresholding this difference to obtain a binary image 3) computing the horizontal projection (row-wise sum) of this image 4) thresholding this projection to extract horizontal bands where moving objects are likely to be found 5) computing the vertical projection (column-wise sum) on each band and 6) applying a peak detector to the projections to define the position of each moving object in the band.

The encoding of this algorithm in CAPH appears in Fig. 8. It consist of five sections. The first section is used for defining type abbreviations. Native types in CAPH include signed and unsigned sized integers. The second section defines global constants (global functions can also be defined here). In the third section, the behavior of all the actors involved in the network is defined. Here, due to space limitations (and because several examples have already been given), the text of the descriptions has been omitted. The fourth stream declaration section is where network input and output streams are defined. In this particular case, the input stream is read from a camera and the output stream is written to a display. Finally, the last section gives the functional equations defining the network.

The code involves eight different actors. The asub actor computes the absolute value of the difference between two frames. The clf is a frame delay operator. The thr actor has been described in Section 4.2. The hproj actor computes the horizontal projection of an image. The vwin actor extracts an horizontal band from an image according to the profile of the thresholded horizontal projection. The vproj computes the vertical projection of an image (the band is represented exactly as an image). The peaks actor analyzes the vertical projection and computes a list of pairs, each pair giving the position of two consecutive peaks. Finally, the win actor uses the positions computed by the vwin and peaks actors to display a frame around the detected objects.

The corresponding dataflow graph, obtained with the graph visualizer is depicted in Fig. 9 (square boxes represent actors, triangles input and output and edges are labeled with the type of corresponding channel).

\footnote{Dedicated VHDL processes, transparent to the programmer, handles the insertion (resp. removal) of control tokens after (resp. before) the image date is read from (resp. written to) camera (resp. display).}
Fig. 8. Source code for the motion detection application (excerpt)

**VHDL Implementation.** Our current target platform is a smart-camera integrating an FPGA board, an image sensing device and communication board. It is fully described in [10]. The FPGA board consists of one FPGA (a Stratix EP1S60), five 1MB SRAM banks and one 64MB SDRAM block. Two dedicated VHDL processes provide interfacing of the generated actor network to the physical I/O devices (camera and display). The VHDL code produced by the CAPH compiler is compiled and downloaded to the FPGA using the Altera Quartus toolset. Small FIFOs are implemented using logic elements. Mid-sized ones (for line delays for example) use the on-chip SRAM memory banks of the FPGA. Frame delay FIFOs (such as the one required by the d1f operator) use the SDRAM. The size of each FIFO is currently estimated using the SystemC back-end, by running an instrumented version of the generated code in which the occupation of the FIFOs is monitored at run-time. In many cases, it should be possible to obtain these sizes at compile-time by performing a static analysis of each actor behavior. In the general case, this approach raises challenging problems, however, and has not been investigated yet. In this particular example, four-places FIFOs are sufficient on all channels except on wire W7 (see Fig. 9), where a FIFO with a depth of at least one line must be inserted (because the value of the horizontal projection for a given line is only available at the end of this line) and on wire W11, where a FIFO with a depth of one frame must be inserted (because the positions of the bounding frames for one frame is only known at the end of this frame)8.

The application operates on the fly on video streams of 512 × 512 × 8 bit images at 15 FPS, with a clock frequency of 150 MHz. It uses 3550 logic elements (6 %), 17 kbits of SRAM and 512 kB of SDRAM.

8. RELATED WORK

The approach followed in CAPH bears similarities with that adopted by other stream-processing and or actor-based languages such as CAL [11] and Canals [12]. All of them share the idea of a network of computational units exchanging data through unidirectional channels following the basic dataflow model of computation. Computational units are called actors in CAL and kernels in Canals. The differences

---

--- Type declarations

```vhd
type byte = unsigned<8>
type bit = unsigned<1>
```

--- Constants (thresholds)

```vhd
const k1 = 30 -- for binary image
const k2 = 1200 -- for hor. projection
const k3 = 900 -- for vert. projection
```

--- Actor declarations

```vhd
actor asub ()
in (a:byte dc, b:byte dc)
out (c: byte dc)
...
actor d1f () ...
actor thr (t:byte) ...
actor hproj () ....
actor vwin () ...
actor vproj () ...
actor peaks (t:byte) ...
actor win () ...
```

--- I/O streams

```vhd
stream i : byte dc from "camera:0"
stream o : byte dc to "display:0"
```

--- Network definitions

```vhd
ndef diff_im = asub (i, d1f i)
ndef bin_im = thr k1 diff_im
ndef hp = thr k2 (hproj bin_im)
ndef hband = vwin (hp, bin_im)
ndef vp = vproj hband
ndef o = win (peaks k3 vp, i)
```

--- Fig. 9. Dataflow graph for the motion detection application

--- It is possible to get rid of this FIFO by inserting a s1f (skip one frame) operator on the corresponding wire. In this case, the bounding boxes computed on frame number i are actually displayed on frame i + 1, which is acceptable if the objects don’t move too quickly.
with CAPH mainly come from the scheduling of execution in actors on the one hand and the syntax and semantics of the network language on the other hand.

Both CAL and Canals allow complex execution scheduling to be specified for actors. For example, CAL provides constructs for expressing guards, priorities or even finite state machines based scheduling within each actor. Canals comes with a sub-language to define scheduling of kernels within the network. By contrast, scheduling is kept much simpler in CAPH since it is entirely specified by the pattern matching rule-based mechanism. This has been made possible by allowing control tokens (namely the '<' and '>' tokens) within the streams exchanged between actors. This approach in turn greatly simplifies the generation of HDL code, which basically boils downs to finite state machines, easily encoded in VHDL.

Both CAL and Canals use a dedicated network language (NL) to describe the network of actors. The abstraction level of these languages is low: the programmer must manually "wire" the network by explicitly listing all the actors and their connexions, a tedious and error-prone task. The network language embedded in CAPH allows implicit description of network by means of functional expression and naturally supports higher-order constructs.

Finally, let us note that the concept of match-based transitions used in CAPH for describing actor behavior has been borrowed from the Hume [13] language, in which it is used to describe the behavior of asynchronous boxes connected by wires. But the semantics of Hume and CAPH are different. In Hume boxes are stateless, wires provide memorization for a single token and the execution model is based on the concept of global cycles.

9. CONCLUSION

We have introduced CAPH, a domain-specific language for programming stream-processing applications on FPGAs. The presentation adopted here is deliberately informal. Its goal is to give an idea of the motivations, basic principles and capabilities of the language. A language reference manual, including the full syntax and formal semantics, is in preparation.

Preliminary examples, such as the one described in this paper, show that efficient implementations of reasonably complex applications can be obtained with a language whose abstraction level is significantly higher than that of traditional HDL languages such as VHDL or Verilog.

Work is currently undergoing in three directions. First is improving the compiler (for now, the set of expressions which can be used in the right-hand sides of the rule is limited to arithmetical and logical operations, 1D-array access, conditional and let: local declarations). Second is applying static WCET analysis techniques to actor behaviors in order to statically estimate the size of FIFO channels (instead of using run-time estimated values). Third is to tackle larger and more complex applications.

10. REFERENCES