CAPH : A high-level actor-based language for programming FPGAs

J. Sérot, F. Berry, S. Ahmed
Institut Pascal, UMR 6602 Université Blaise Pascal / CNRS
Clermont-Ferrand, France

WASC 2012

Apr 5-5 2012, Clermont-Ferrand

Context and motivations

- Generalisation of heterogenous embedded systems
  - hw +sw (typically : CPU + FPGAs)
- Large opportunities for performance improvements (massive parallelism, close-to-sensor processing, ...)

Context and motivations

- This raises challenging issues for the designer
  - Programming concepts, techniques and tools are still very different for software and hardware
  - The very notion of "program" is quite different for a software programmer and a hardware "designer" !

The big issue

- Most software programmers find it hard to make use of hardware reconfigurable ("programmable") devices
- Some reasons are "technical" (tools, ...)
- ... but the key issue has to do with the respective programming models

Programming models

- In most of (in not all) software programming models, time is implicit
  - this is possible because the model is *sequential*
  - Ex : `x := x+1; x := x*2;`

    it does not really matter *when* (at what date), the second instruction is executed; the only thing that matters is that it is carried out *after* the first one)
- Variation in the programming model (functional, object-oriented, ...) does not fundamentally change this

Programming models

- By contrast, in hardware "programming" models, time (and the related concept of *synchronisation*) is generally explicit
- Ex :VHDL :

```
process(clk) begin q <= '0'; r <= q+1;... end;
```

- Moreover, separation between data and control signals
  - in particular for systems operating "on the fly" (*stream processing applications*)
  - does not exist in software programming !
- **This is what makes hardware programming hard for software programmers !**

The big issue

How to make "hardware programming" acceptable for "software" programmers ?

The big issue : "" "standard" answer

- Make hardware description languages closer to "software" programming languages

➡ C-like HDLs (SystemC, HandelC, ...)

C-like HDLs ?

... are not the panacea !

- Some concepts of the sequential / imperative programming model do no map easily/efficiently into hardware (ex: random memory access)
- Some constructs of the "source" language must be avoided
- Ultimately requires knowledge on hardware programming...

  .... which is precisely what we want to avoid !
- Require complex (and hence hard to prove correct) transformations to be implemented

Why C-like HDLs fail

- The gap between the specification and the implementation is too large
  - esp. : control signals
    - making them explicit at the specification level breaks the abstraction barrier
    - ... but inferring them from a high-level C description is hard !

Reducing the gap

- What is needed is an adequate programming model
  - Offering an homogeneous view of control and data values / signals
  - ... thus supporting "software oriented" descriptions of stream-processing applications
  - ... but also leading to efficient hardware implementations

Finding an adequate programming model

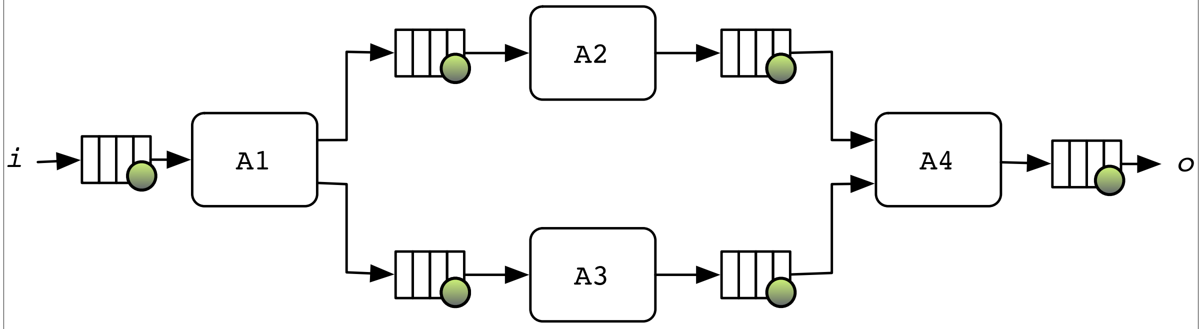
- Pragmatic (bottom-up) approach !
- Q : What can be easily / efficiently implemented in hardware ?
- A (partial) :
  - combinational logic
  - FSMs
  - FIFO based communications
- Can we base a "software-programmer-friendly" programming language on this ?

The basic building blocks

- Combinational blocks can implement all *pure* (state-less) computations
- FIFO-based communication fits nicely within data-flow / actor based programming models
  - these models are highly intuitive (and familiar to programmers - esp. in DSP area)
  - control signals can be *embedded* as special data tokens

Dataflow model

- Very old idea !
- An application = a collection of computing units (*actors*) exchanging tokens through unidirectional, *buffered* links (*FIFOs*)
- For each actor, a set of firing rules specifies when it consumes input tokens and produces output tokens

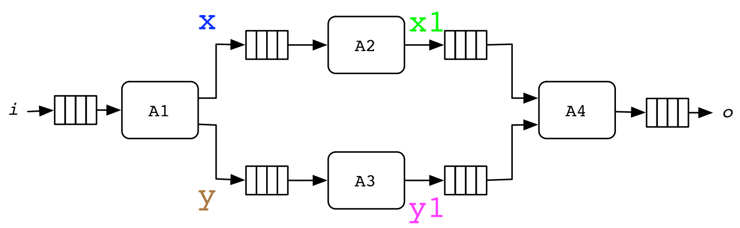


From to concepts to the language

- Need a formalism to describe / specify
  - the network topology
  - the behavior of actors
  - what tokens contain / represent

Describing networks

➔ Textual (functional) description



```
net (x,y) = A1 i
net x1 = A2 x
net y1 = A3 y
net o = A4 (x1, y1)
```

or

```
net (x,y) = A1 i
net o = A4 (A2 x, A3 y)
```

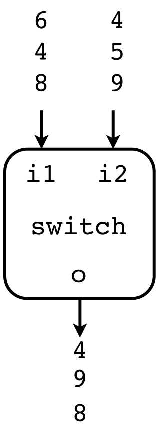
```
net diamond f g h k x =
  let x,y = f i in
  k (g x, h y)
net o = diamond A1 A2 A3 A4 i
```

➢ Consistency can be checked using type-checking
➢ Reusable graph patterns can be encapsulated as *higher-order functions*

Describing actor behavior

➔ Generalized Finite State Machines

- Borrowed from the *Hume* language
- Behavior described a a set of transition rules
- Activation of rules based on pattern-matching



```
actor switch
in  (i1 : int,
     i2 : int)
out (o : int)
var s : (Left,Right) = Left
rules (s,i1,i2) → (o,s)
| Left, v, _ → v, Right
| Right, _, v → v, Left
```

I/Os

Local var

Rule format

Transition rules

Generalized FSMs

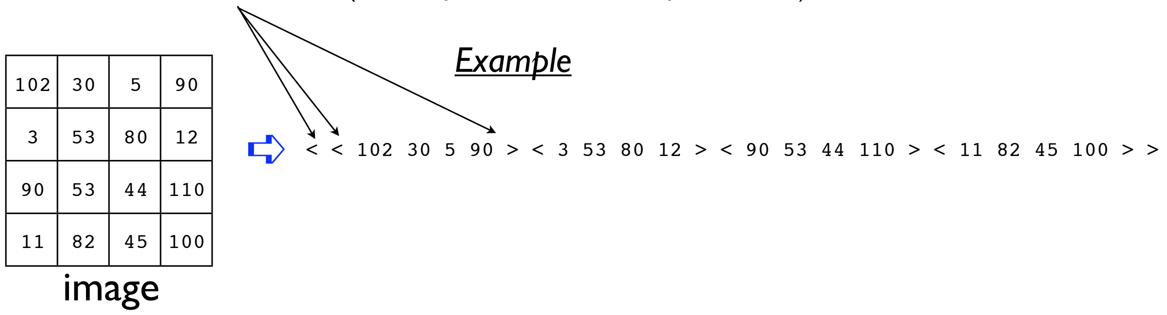
- Cleanly separate computation from communication concerns
- Trade-off between expressivity and predictabily governed by the computation language used for describing actions (from purely combinational fns to recursive or higher-order fns) [inspired from Hume approach]

Data representation

- Key issue for stream-processing applications
- Synchronisation problems are *partly* solved by using FIFOs to connect actors ...
- ... but we still need to represent the structure of the processed data
    - ex: detect *start/end_of_frame* and *start/end_of_line* in a stream of images

Data representation

- Two categories of tokens
    - data tokens (integers, booleans, pixels, ...)
    - control tokens (*start_of_structure, end_of_structure)*



102	30	5	90
3	53	80	12
90	53	44	110
11	82	45	100

```
< < 102 30 5 90 > < 3 53 80 12 > < 90 53 44 110 > < 11 82 45 100 > >
```

➢ Can be used to represent arbitrarily structured data (remember Lisp ?)

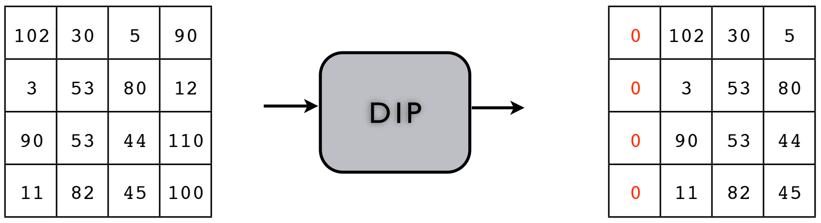
➢ No global control / synchronisation signals needed

➢ Naturally supports pipelined execution

Data representation

➢ Functional representation of temporal operation

Example : 1-pixel delay



102	30	5	90
3	53	80	12
90	53	44	110
11	82	45	100

0	102	30	5
0	3	53	80
0	90	53	44
0	11	82	45

< < 102 30 5 90 > < 3 53 80 12 > < 90 53 44 110 > < 11 82 45 100 > >

⇩

< < 0 102 30 5 > < 0 3 53 80 > < 0 90 53 44 > < 0 11 82 45 > >

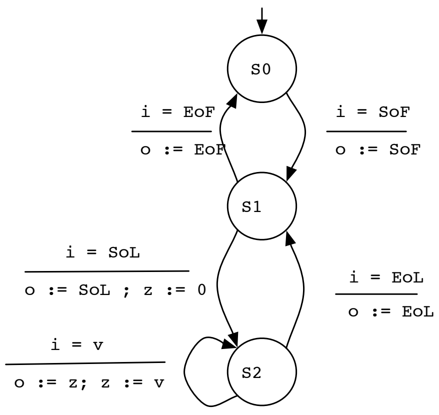
Data representation and actor behavior

< < 102 30 5 90 > < 3 53 80 12 > < 90 53 44 110 > < 11 82 45 100 > >

⇩ **d1p**

< < 0 102 30 5 > < 0 3 53 80 > < 0 90 53 44 > < 0 11 82 45 > >

```
actor d1p
  in (i:signed<8> dc)
  out (c:signed<8> dc)
var s : {S0,S1,S2} = S0
var z : signed<8>
rules (s, i, z) -> (s, o,z)
| (S0,     SoF, _) -> (S1,     SoF, _)
| (S1,     EoF, _) -> (S0,     EoF, _)
| (S1,     SoL, _) -> (S2,     SoL, 0)
| (S2, Data v, z) -> (S2, Data z, v)
| (S2,     EoL, _) -> (S1,     SoL, _)
```



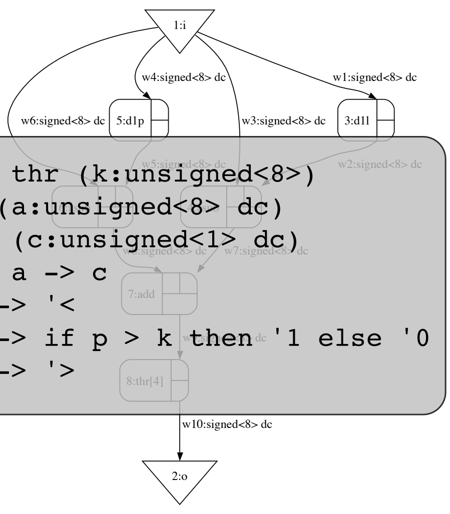
A sample *Caph* program

```
function f_abs x =
  if x < 0 then 0-x else x
  : signed<8> -> signed<8>;

constant threshold = 40;
```

Global values

```
actor d1p () ...

actor d1l () ...

actor add () ...

actor asub () ...

actor thr (t:signed<8>) ...
```

Actors



```
actor thr (k:unsigned<8>)
  in (a:unsigned<8> dc)
  out (c:unsigned<1> dc)
rules a -> c
| '< -> '<
| 'p -> if p > k then '1 else '0
| '> -> '>
;
```

I/Os

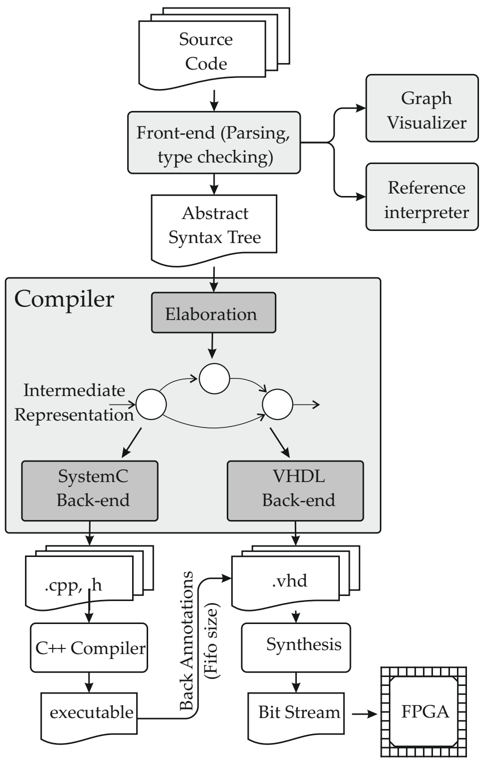
```
stream i:signed<8> dc from "dev:cam0";
stream o:signed<8> dc to "dev:mon1";
```

```
net g = add (asub(i, d1p i), asub(i, d1l i));
net o = thr [threshold] g;
```

Network description

The *Caph* toolset

- Graph visualizer : .dot format
- Reference interpreter :
  - based on the fully formalized semantics
  - tracing, profiling and debugging
- Compiler :
  - elaboration of a target-independant IR
  - specialized backends (SystemC, VHDL)



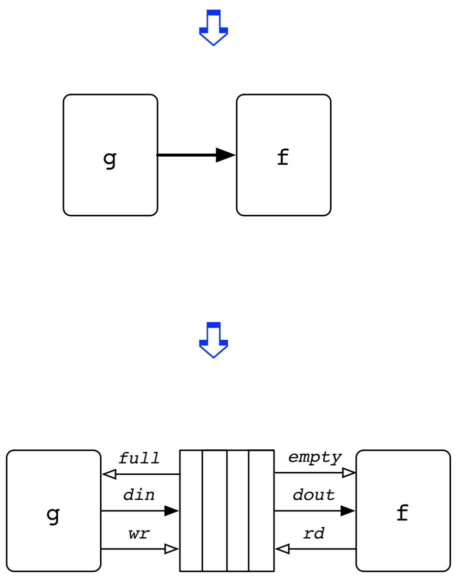
Elaboration

- Three steps
  1. network generation
  2. actor translation
  3. SystemC/VHDL transcription
- Only a quick overview here (see papers & LRM)

1. Network generation

- Using *abstract interpretation* of the network description, viewed as a *functional program*
- Each application of a function bound to an actor inserts an instance of this actor into the graph
- Each functional dependency inserts a *channel*
  - channels are then instanciated as FIFOs

```
net y = f ( g x)
```

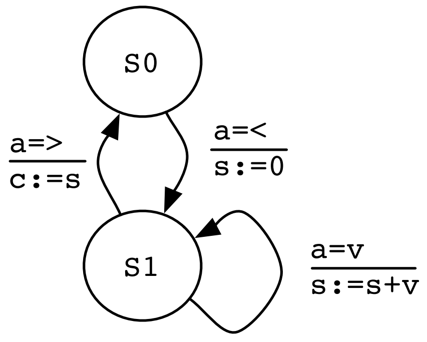


II. Actor translation

Set of transition rules → FSM + operations (FSMD)

Example : **suml : < 1 2 3 > = 6**

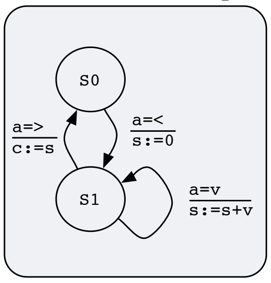
```
actor suml ()
  in (a: int dc)
 out (c: int)
 var st: {S0,S1}=S0
 var s : int
 rules st,a,s-> st,c,s
   S0, '<, _ ➔ S1, _, 0
 | S1, 'v, s ➔ S1, _, s+v
 | S1, '>, s ➔ S0, s, _
```



III. VHDL Transcription

Example

```
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,_
```



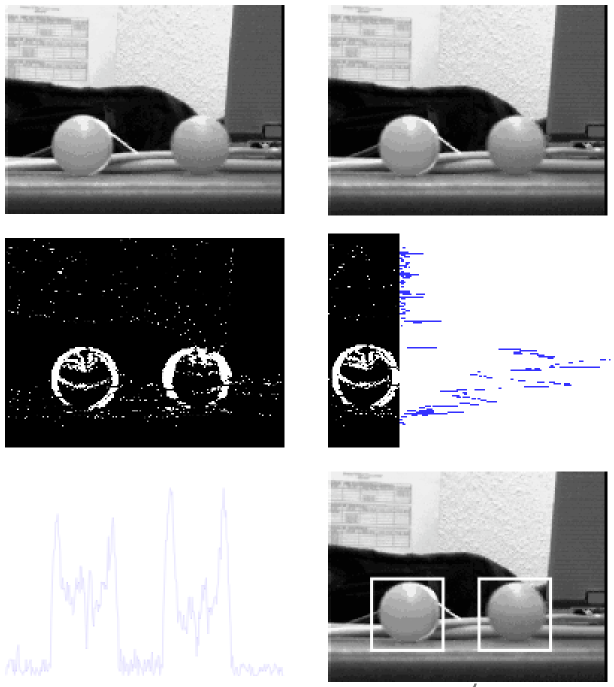
```
entity sum_act is
   port (
     a_empty: in std_logic;
     a: in std_logic_vector(9 downto 0);
     a_rd: out std_logic;
     c_full: in std_logic;
     c: out std_logic_vector(15 downto 0);
     c_wr: out std_logic;
     clock: in std_logic;
     reset: in std_logic
     );
end sum_act;

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;
          if a_empty='0' and is_eos(a) then
            a_rd <= '1'; c := s;
            c_wr <= '1'; st := S12;
            end if;
        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;
```

A realistic example

Real-time tracking of moving objects in a digital video stream

1. thresholding a difference image btw two successive frames to get a binary image
2. thresholding the horizontal projection to get horizontal bands containing moving objects
3. computing and analysing vertical projection on each band to get position of moving objects

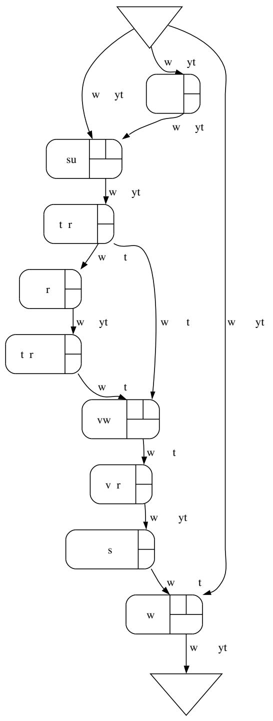


A realistic example

Source code (extr.)

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

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

actor asub () ...
actor d1f () ...
actor thr (t:byte) ...
actor hproj () ....
actor vwin () ...
actor vproj () ...
actor peaks (t:byte) ...
actor win () ...

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

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

Dataflow network



A realistic example

- Implemented on a smart-camera (!) platform embedding a *Stratix EP1S60* FPGA
- Synthesis of VHDL code produced by *Caph* compiler using the *Quartus* toolset
- Interfacing to i/o devices via dedicated VHDL processes
- FIFO are implemented with LEs, on-chip or external RAM banks depending on their size
  - this size is currently estimated by running an *profiled* version of the code generated by the SystemC backend

A realistic example

- FPGA utilisation :
  - 3550 LEs (6%)
  - 17 kbits SRAM
  - 512 kB externam RAM (one-frame delay)
- Fmax = 150 MHz
- Correctly processes streams of 512 x 512 x 8 bits images at 15 FPS

Conclusion

- A new domain-specific language for programming stream-processing applications on FPGAs
- Substantial increase in abstraction level compared to VHDL/Verilog
  - .... without significant performance penalty
- Fully formalized approach (see LRM)
  - complete formal semantics for the language
  - formalized and tractable compilation path
- Toolset and manual available at

  http://dream.univ-bpclermont.fr/index.php/en/caph-v-13.html

Conclusion

- Still a prototype !
- Work under progress :

  - breaking complex computations into multiple clock cycles (important issue !)

  - static (compile-time) estimation of FIFO sizes

  - more complex applications (ex : MPEG decoder/encoder)
- Test and feedback welcome !

One last thing ...

- What does CAPH stands for ?
  - Caph Ain’t just Plain HDL
  - β Cassiopeia