Modeling Reconfigurable Systolic Arrays for Computing Algebraic Operations via Rewriting-Logic

Mauricio Ayala-Rincón[1,4], Ricardo P. Jacobi[2,4], Carlos H. Llanos[3,4] and Reiner W. Hartenstein[5,6]

[1] Departamento de Matemática, ayala@mat.unb.br
[2] Departamento de Ciência da Computação, rjacobi@cic.unb.br
[3] Departamento de Engenharia Mecânica, llanos@unb.br
[4] Universidade de Brasília
[5] Fachbereich Informatik (CS department), hartenst@rhhk.uni-kl.de,
[6] Kaiserslautern University of Technology

Abstract

Systolic arrays provide a large amount of parallelism. However, their applicability is restricted to a small set of computational problems due to their lack of flexibility. This limitation can be circumvented by using reconfigurable systolic arrays, where the node operations and interconnections can be redefined even at run time. In this context, several alternative systolic architectures can be explored and powerful tools are needed to model and evaluate them. Well-Known rewriting-logic environments such as ELAN and Maude can be used to specify, simulate and even synthesize complex application specific digital system. In this paper we propose the use of rewriting-logic to model and evaluate reconfigurable systolic architectures.

Keywords: *Term rewriting Systems, Rewriting-Logic, Reconfigurable Systolic arrays,.*

1. Introduction

The widespread popularization of mobile computing and wireless communication systems fostered the research on new architectures to efficiently deal with communications issues in hardware constrained platforms like PDAs, mobile phones and pagers, for instance. Some tasks such as data compression, encoding and decoding are better implemented through dedicated hardware modules than using standard general purpose processors (GPP). However, the exploding costs of integrated circuits fabrics associated with shorter devices lifetimes makes the design of ASIC (Application Specific Integrated Circuit) a very expensive alternative. The growing capacity of Field Programmable Gate Arrays (FPGA) and the possibility of reconfiguring them to implement different hardware architectures makes it a good solution to this rapid changing wireless market. An FPGA may be configured to implement a cipher algorithm at one moment and can be later reconfigured to implement a data compressing algorithm. This flexibility opens a wide range of architectural alternatives to implement algorithms directly in hardware. In this context, it is very important to provide methods and tools to rapidly model and evaluate different hardware architectures to implement a given algorithm.

In this paper we propose the use of rewriting systems to model and evaluate reconfigurable systolic hardware architectures. After the seminal work of Knuth-Bendix about the *completion* of algebraic equational specifications, which allows for the automatic generation of a rewrite-based theorem prover for the equational reduct of the subjacent treated theories [KnBe1970], rewriting has been successfully applied into different areas of research in computer science as an abstract formalism for assisting the simulation, verification and deduction of complex computational objects and processes. In particular, in the context of computer architectures, rewriting theory has been applied as a tool for reasoning about hardware design.

To review only a reduced set of different approaches in this direction, we find of great interest the work of Kapur who has used his well-known *Rewriting Rule Laboratory - RRL* (the first successful prover assistant based on rewriting) for verifying arithmetic circuits [KaSu2000, Ka2000, KaSu1997] as well as Arvind's group that treated the implementation of processors over simple architectures [ShAr1998a,ShAr1998b,ArSh1999], the rewrite-based description and synthesis of simple logical digital circuits [HoAr1999] and the description of cache protocols over memory systems [RuShAr1999,ArStSh2001]. Also we have contributed in this field by showing how rewriting

Figure 6 shows one additional rule created for the reconfiguration of a processor called conf. It simply changes the contents of the constant part of each MAC (in our case by the vector (1.0,0)). Observe that with the pure rewriting based paradigm this rule applies infinitely. Thus for controlling its application, we define a logical strategy, called withconf.

```
[conf]

[1,port(vp11,true),port(0,true),reg(vr11,true),reg(vr12,true),c1]
[2,port(vp21,true),port(vp22,true),reg(vr21,true),reg(vr22,true),c2]
[3,port(vp31,true),port(vp32,true),reg(vr31,true),reg(vr32,true),c3]
      (d1.l1 d2.l2 d3.l3) >
    =>
  <[1,port(vp11,true),port(0,true),reg(vr11,true),reg(vr12,true),1]
    [2,port(vp21,true),port(vp22,true),reg(vr21,true),reg(vr22,true),0]
    [3,port(vp31,true),port(vp32,true),reg(vr31,true),reg(vr32,true),0]
         (d1.l1 d2.l2 d3.l3) >
  end
strategies for Proc
   implicit
      [] withconf    => conf; normalise(sole) end
      [] simple      => normalise(sole) end
end
```

Figure 6: *conf* Rule for Reconfiguration

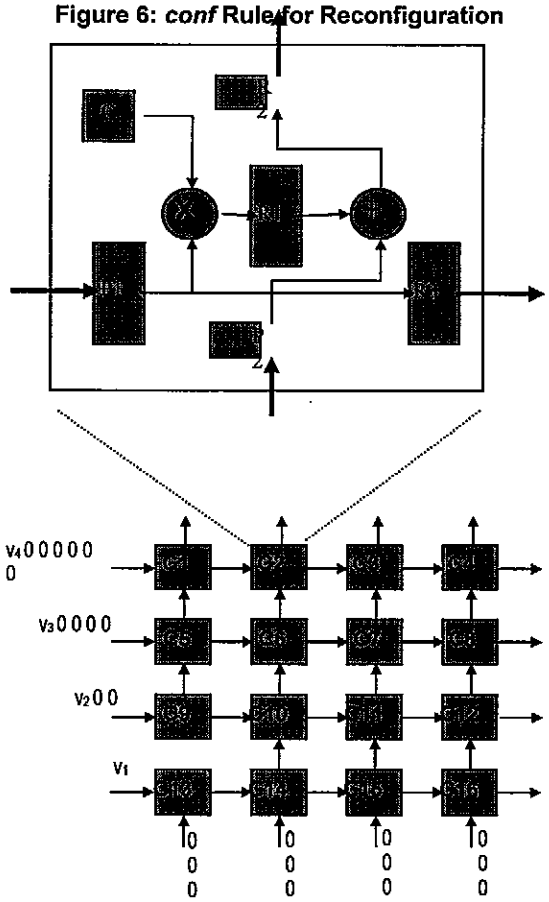

Figure 7: Systolic Matrix-vector multiplication

withconf simply allows for the execution of one-step of reduction with the rule conf (the first reconfiguration stage) and a subsequent normalization with the rule sole (the second processor execution stage).

In the sequel, we will show how a systolic array for 4x4 matrix multiplication (see Figure 7) has been implemented in ELAN.

The description of the processor and its components are given in the Figure 8. But its direct specification implies the use of an excessive number of variables for a sole rule, as in our previous example, that is not directly supported by ELAN. In fact, we will need different variables for the two ports, three registers and the constant belonging to each MAC, which gives a total of 96 variables; additionally, we will need 16 variables for describing the two 4x data streams. This could be done by enlarging the capacity of ELAN for dealing with variables before compiling the system. But a better solution is to split the cycle defining independent rewriting rules, to be applied under a reasonable strategy, for simulating the internal process into each MAC component and the propagation of data between each component to their North and East connected MACs.

```
operators global
 @                          : ( int ) Const;
 p(@)                       : ( int ) Port;
 r(@)                       : ( int ) Reg;
 '['@,@,@,@,@,@,@']' : ( int Port Port Reg Reg Reg Const
                            ) MAC;
 '<' @
    @ @ @ @
    @ @ @ @
    @ @ @ @
    @ @ @ @
    @ '>
 : ( DataString
            MAC MAC MAC MAC   // MACs 13 14 15 16
            MAC MAC MAC MAC   // MACs 09 10 11 11
            MAC MAC MAC MAC   // MACs 05 06 07 08
            MAC MAC MAC MAC   // MACs 01 02 03 04
            DataString    ) Proc;
 ( @ @ @ @ )  : ( list[Data] list[Data] list[Data]  list[Data] )
    DataString;
 @              : ( int ) Data;
end
```

Figure 8: a 4×4 Systolic array Description

We define a rule for each of the sixteen components, which propagates the contents into their registers two and three to their North and East connected components, respectively.

As consequence of the form in which data is transferred in the processor, these sixteen rules should be applied from the right to the left and top-down in order to complete a whole cycle of execution.

All these rules are very similar and a selected group of them is presented in the Figure 9.

Observe that the rules for the South (mac01, mac02, mac03, mac04) and West (mac01, mac05, mac09, mac13) boundary components of the processor charge the data (dS and dW) from the

head of the corresponding list of the data stream (IS1, IS2, IS3, IS4 and IW1, IW2, IW3 and IW4).

```
rules for Proc
  m01,m02,m03,m04,m05,m06,m07,m08: MAC; // 1-8 MACs
  m09,m10,m11,m12,m13,m14,m15,m16:MAC; //9-16 MACs
  dW, dS                  : int; // data East and South
  IW1,IW2,IW3,IW4,IS1,IS2,IS3,IS4:list[Data]; // West and South
  r1,r2, r3,rN1,rN2,rN3   : int;  // Central North and
  rE1,rE2,rE3             : int;  // East registers 1,2,3
  p1,p2,pN1,pN2,pE1,pE2: int; //Central,North and East ports
  c,cE,cN                 : int;
global
[mac16]
<  (IW1 IW2 IW3 IW4)
   m13 m14 m15 [16,p(p1),p(p2),r(r1),r(r2),r(r3),c ]
   m09 m10 m11 m12
   m05 m06 m07 m08
   m01 m02 m03 m04
   (IS1 IS2 IS3 IS4) > =>
<  (IW1 IW2 IW3 IW4)
   m13 m14 m15 [16,p(p1),p(p2),r(p1*c),r(r1+p2),r(p1),c ]
   m09 m10 m11 m12
   m05 m06 m07 m08
   m01 m02 m03 m04
   (IS1 IS2 IS3 IS4) >
end   ...
[mac11]
<  (IW1 IW2 IW3 IW4)
   m13 m14 [15,p(pN1),p(pN2),r(rN1),r(rN2),r(rN3),cN ] m16
   m09 m10 [11,p(p1),p(p2),r(r1),r(r2),r(r3),c ]
             [12,p(pE1),p(pE2),r(rE1),r(rE2),r(rE3),cE ]
   m05 m06 m07 m08
   m01 m02 m03 m04
   (IS1 IS2 IS3 IS4) > =>
<  (IW1 IW2 IW3 IW4)
   m13 m14 [15,p(pN1),p(r2),r(rN1),r(rN2),r(rN3),cN ] m16
   m09 m10 [11,p(p1),p(p2),r(p1*c),r(r1+p2),r(p1),c ]
             [12,p(r3),p(pE2),r(rE1),r(rE2),r(rE3),cE ]
   m05 m06 m07 m08
   m01 m02 m03 m04
   (IS1 IS2 IS3 IS4) >
end   ...
...
...
 [mac01]
<  (dW.IW1 IW2 IW3 IW4)
   m13 m14 m15  m16
   m09 m10 m11  m12
   [05,p(pN1),p(pN2),r(rN1),r(rN2),r(rN3),cN ] m06 m07 m08
   [01,p(p1),p(p2),r(r1),r(r2),r(r3),c ]
   [02, p(pE1),p(pE2),r(rE1),r(rE2),r(rE3),cE]m03 m04
  (dS.IS1) IS2 IS3 IS$) > =>
<  (IE1 IE2 IE3 IE4)
   m13 m14 m15 m16
   m09 m10 m11 m12
   [05, p(pN1),p(r2) r(rN1) r(rN2) r(rN3), cN] m06 m07 m08
   [01, p(dW), p(dS),r(p1*c), r((rE2), r(rE3), cE] m03 m04
   (IS1 IS2 IS3 IS4)>
end
end
```

Figure 9: a selected set of rules for matrix-vector multiplier

Also observe that the rules for MACs in the North (mac13, mac14, mac15, mac16) and East (mac04, mac08, mac12, mac16) boundaries of the processor only transfer data to the East and North corresponding boundary components; except, of course, for mac16. Thus different orderings for applying these rules completing a whole cycle of the processor are possible. For instance, take the ordering mac16, mac12, mac08, mac04, mac15, mac11, mac07, mac03, mac14, mac13, mac10, mac09, mac06, mac05, mac02, mac01.

In the Figure 10 we present a possible strategy called onecycle which defines an(other) ordering of application of these rules for completing a sole cycle of the processor. For completing the simulation of execution with this simple processor, one should define a normalization via this strategy: normalise(onecicle).

In this rewriting-logical environment, our specification could be easily modified allowing the interpretation of parts of the processors as reconfigurable components.

```
Strategies for Proc
 implicit
  []  onecycle =>
        mac16;mac15;mac14;mac13;
        mac12;mac11;mac10;mac09;
        mac08;mac07;mac06;mac05;
        mac04;mac03;mac02;mac01
      end
end
```

Figure 10: onecycle strategy for rule application

At first glance, one could look at the constants of the 16 MACs as a reconfigurable component. In this way the processor can be adapted to be either a 4-vector versus 4x4-matrix multiplier or vice-versa and the 4x4-matrix may be modified to represent, for example, either the identity or the F_4 matrix of the Discrete Fourier Transform.

The last is specified by an additional strategy, presented in the Figure 11 which before to the simulation of the normalization executes a rewrite rule F4 working as an operator over processors.

```
reconfF4 => F4; normalise(mac16;mac15;mac14;mac13;
                          mac12;mac11;mac10;mac09;
                          mac08;mac07;mac06;mac05;
                          mac04;mac03;mac02;mac01)
end
```

Figure 11: strategy working over processors

The rule F4 transforms any given state of the processor into another where the constants are replaced with the corresponding powers of a primitive complex 4-root of the unity of F_4 (either i or -i) as illustrated in the Figure 12.

In this specification the components of each MAC have been divided into the fixed ones and the reconfigurable constant [fxnn cnn]. This simple idea can be straightforwardly extended for other kind of MACs, where other components are considered reconfigurable as we will see in the next section.

```
[F4]
< dstreamEast
   [fx13 c13] [fx14 c14] [fx15 c15]  [fx16 c16]
   [fx09 c09] [fx10 c10] [fx11 c11]  [fx12 c12]
   [fx05 c05] [fx06 c06] [fx07 c07]  [fx08 c08]
   [fx01 c01] [fx02 c02] [fx03 c03]  [fx04 c04]
   dstreamSouth > =>
<  dstreamEast
   [fx13 i^0]  [fx14 i^3]  [fx15 i^6]  [fx16 i^9]
   [fx09 i^0]  [fx10 i^2]  [fx11 i^4]  [fx12 i^6]
   [fx05 i^0]  [fx06 i^1]  [fx07 i^2]  [fx08 i^3]
   [fx01 i^0]  [fx02 i^0]  [fx03 i^0]  [fx04 i^0]
   dstreamSouth >
end
```

Figure 12: rule for FFT Transformer

4. Modeling more general reconfigurable systems

4.1 Dynamic Reconfiguration

In order to illustrate the possibilities of applying rewriting-logic environments such as ELAN and Maude in the specification and subsequent simulation of idealized reconfigurable systems, we extend the previous approach, where we have considered only the constant components of the MACs of our systolic arrays as reconfigurable parts, to other components such as operators and address registers.

This is exemplified by modeling the Fast Fourier Transform (FFT) in a compact array of MACs which uses dynamic reconfigurations. That is, this array changes its configuration at run time. This is illustrated for the calculation of F_8. Classical circuits for F_n are well-known and can be found in popular textbooks on algorithms [CLRS2001].

Our 8-array is founded on these circuits and its (operational semantics and) correctness is based on

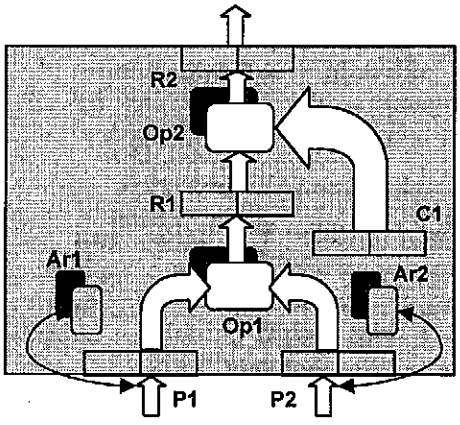

Figure 13: MAC architecture for FFT

the application of an adequate reconfiguration of the operators, constants and address registers alternatively to the execution of the whole operations over the eight MACs that conform the 8-array. For this example, the structure of the MACs is modified as presented in the Figure 13. Here we distinguish between reconfigurable (shadowed) and fixed components. The formers are the two address registers, Ar1 and Ar2; the two operators, Op1 and Op2; and the constant, Ct. The lasts are the two ports, P1 and P2; and the two registers, R1 and R2.

The two registers, the two ports and the constant consist each of two components: the first for the real and the second for the imaginary part of a complex number. The two operators can be reconfigured to be any operation over complex numbers. In particular, for implementing FFT (for F_8) we will use only addition (+), subtraction (-) and multiplication (×) of complex numbers.

The two address registers, Ar1 and Ar2, are used to indicate in each of the eight MACs the origin of the data that should be loaded into the respective ports, P1 and P2. The options for configuration of these address registers are either the input (I) (as input we will supply the coefficients of a given polynomial permuted adequately) or the output (second register R2) of one of the eight MACs (0,1,...,7).

In any reconfiguration the constant part of each MAC is set with arbitrary complex numbers. For implementing FFT, we will set these constants with complex roots of the unity.

The 8-array, jointly with the parameters of its initial reconfiguration, is illustrated in the Figure 14. The initial reconfiguration parameters are given by the sequence:

0: I,I,+,1,*; 1: I,I,+,-1,*; 2: I,I,+,1,*; 3: I,I,+,-1,*;
4: I,I,+,1,*; 5: I,I,+,-1,*; 6: I,I,+,1,*; 7: I,I,+,-1,*;

That means that the MAC 0 receives its inputs from the corresponding external inputs of the whole 8-array; its first operator is configured as addition; its constant as 1; and its second operator as multiplication. Similarly, for the remaining seven MACs. After this reconfiguration, the operations are executed, obtaining in the output second register of each MAC the corresponding coefficient: P0, -P4, P2, -P6, P1, -P5, P3 and -P7, respectively. Then current execution is break and a second reconfiguration applied. This reconfiguration is given by the following sequence:

0: 0,1,+,1,*; 1: 0,1,-,1,*;
2: 2,3,+,i,*; 3: 2,3,-,-1,*;
4: 4,5,+,1,*; 5: 4,5,-,1,*;
6: 6,7,+,i,*; 7: 6,7,-,-1,*;

This means that the first and second address registers for the corresponding ports of the MACs 0

and 1 should be loaded with the outputs of the MACs 0 and 1, being these added in the first MAC and subtracted in the second MAC. In the first and second MAC the constants are 1 and in the third and fourth i and -1. The second operator remains as multiplication. After the second reconfiguration, a complete execution over the MACs proceeds.

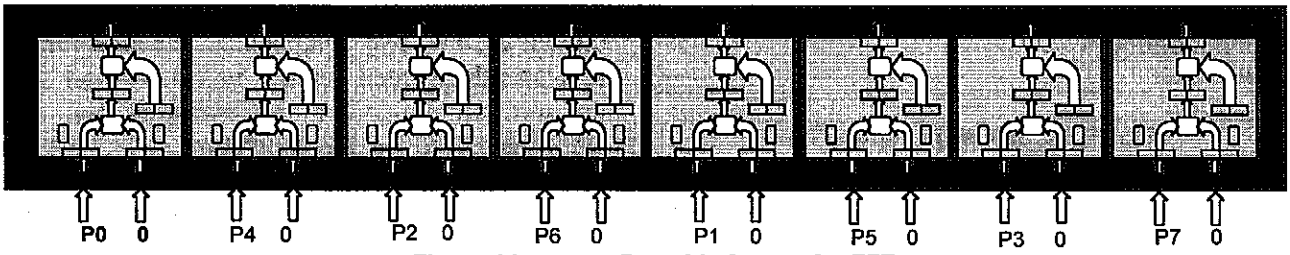

Figure 14: a reconfigurable 8-array for FFT

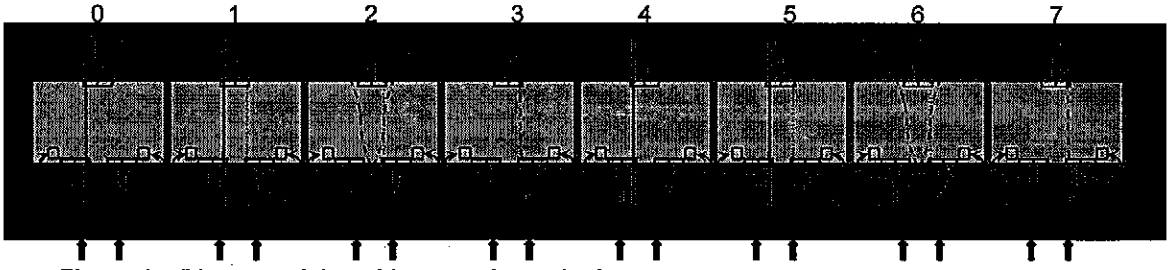

Figure 15. Diagram of the address registers in the FFT implementation

Observe that we will obtain as respective outputs the values: P0-P4, P0+P4, (P2-P6)i, -(P2+P6), P1-P5, P1+P5, (P3-P7)i and -(P3+P7).

The third reconfiguration is given by the sequence:

0: 0,2,+,1,*; 1: 1, 3,+,1,*;
2: 0,2,-,1,*; 3: 1,3,-,1,*;
4: 4,6,+, $(1+i)/\sqrt{2}$,*; 5: 5,7,+,i,*;
6: 4,6,-,$(-1+i)/\sqrt{2}$,*; 7: 5,7,-,-1,*;

The contents of the address registers corresponding with this sequence is illustrated in the Figure 15. Compare with the usual circuits for FFT.
Finally, after a complete execution over the MACs, the 8-array is configured with the following sequence:

0: 0,4,+,1,*; 1: 1,5,+,1,*;
2: 2,6,+,i,*; 3: 3,7,+,1,*;
4: 0,4,-,1,*; 5: 1,5,-,1,*;
6: 2,6,-,i,*; 7: 3,7,-,1,*;

This gives as output F_8 x (P0,...,P7).

Details about the ELAN specification of this reconfigurable 8-array are not given. Description of this specification is similar to these presented in the previous section.

We restrict us to explain how we use logical strategies for simulating this dynamical reconfiguration. The key for correctly simulating this specification is in fact a very simple logical strategy, which alternatively simulates a reconfiguration step and a whole step of simultaneous execution of a cycle in the eight MACs. The former corresponds to a fibreconf step and the last to onecicle. The correct sequence of reconfigurations obtained by applications of steps of the rule fibreconf as well as the end of the process is controlled by the current state of the reconfigurable components of the 8-array (for instance, the current values in the constant components or in the address registers).

Thus we can condense the explanation of the whole ELAN specification and simulation by presenting the very simple strategy:

```
[] processing => normalise(figreconf;onecycle) end
```

With different strategies of (dynamical) reconfiguration the 8-array can be adapted to execute other operations.

4.2 Variable Size Systolic Arrays

A more flexible description will allow the specification of systolic arrays with an arbitrary number of functional units. In this case, the rewriting rules should be defined independently of the array size or topology.

This is exemplified in this section through the modeling of a simple version of the KressArray architecture [HaKrHe95]. It is defined by a matrix of reconfigurable functional units (rDPUs: reconfigurable datapath units) where both, the operations and the interconnections, may be redefined (compare fig. 16 c and d). Its structure is similar to the matrix multiplier array shown in Figure 7. Figure 16 illustrates a KressArray family

design space, which covers a wide variety of reconfigurable connect fabrics: nearest neighbour interconnect (NN) and backbus fabrics (segmented and / or non-segmented). Figure 17 shows a detailed example of NN ports featuring individual path width and individual mode (in, out, or bidirectional).

Mapping C expressions to KressArrays is performed by assigning C operators to the nodes while keeping the corresponding data dependency among them.

One particularity is that a KressArray is a pipe network which is a dataflow like architecture. Coming along with synthesis tools also supporting non-uniform non-regular pipe networks [HHHN00, Na01] (in contrast to classical systolic array synthesis methods accepting only applications with strictly regular data dependencies) the KressArray family is a generalization of the systolic array. In addition to the generalized NN interconnect the KressArray family also provides a backbus (BB)

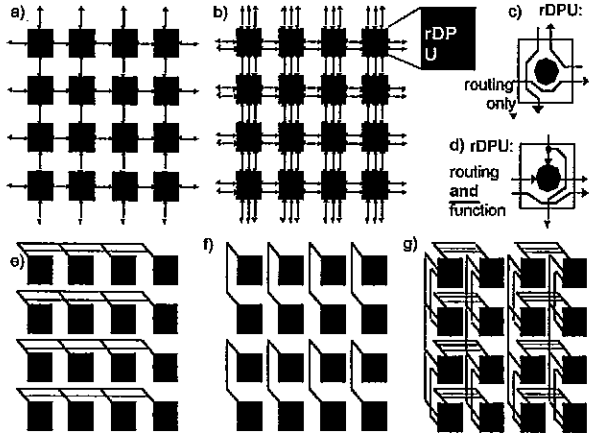

Figure 16: KressArray family design space: (a, b) NN fabrics examples; (e, f, g) backbus (BB) fabrics examples; rDPU configuration: c) routing only configuration, d) routing and function configuration

second level interconnect fabrics with resources like buses and bus segments (for family member examples see fig. 16 e, f, g).

The DPU nodes of both, systolic arrays and rDPAs, may operate in a clocked mode, or asynchronously, where each operations is triggered as soon as data is available at the node inputs. (The latter version of systolic arrays has been usually called wavefront arrays). This asynchronous operation is accomplished by a handshake between interconnected nodes, since each operation may take several clock cycles (multiplication, for instance, is implemented in its typical serial way, through sums and shifts).

Modeling of KressArrays in ELAN takes a different approach due to its particularities. The nodes are stored on a list of arbitrary length. The designer provides the interconnections among nodes by specifying the variables names associated to the registers at the inputs and outputs of the nodes. Thus, if node n_i is connected to node n_j, then the same variable is associated to n_i output and n_j input.

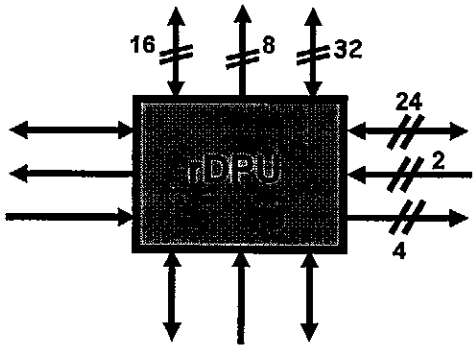

Figure 17: a KressArray family member example illustrating individual port mode and path width

Figure 18 shows an example of a KressArray application description, mapped onto the architecture platform shown in fig. 16 a.

Each rDPU nodes is a kind of rAlu (reconfigurable Arithmetic-Logic Unit). In this simplified version, each rDPU has two registers in the inputs and one register at the output. Each register is defined by the variable it holds, its value and a flag that indicates if the data it holds is valid. The flag models the signal used for handshake in hardware. Rule assign() is used by the designer to provide values to input variables. The Systole strategy normalizes the rule taking the first result (basic ELAN strategy first one) produced by dpu() rule, which is presented in Figure 19.

Rule exec() computes the output value of each rAlu. This rule simply applies the operation specified in the node to its inputs and update the output register, setting its ready flag to true. The rules update() and propagate() traverse the rAlu list updating the input of the nodes connected to the node that was processed.

The fact that ELAN process sequentially the input is not an issue, due to the data flow nature of this array. After the primary (external) inputs are defined by the user, the array is simulated iteratively until all nodes have processed their respective inputs. When a node computes a new value, it is propagated through the array to all registers that depends on that variable through the rules update() and propagate() cited above. The

node's control flag is set to true, indicating that the register data is ready to be used by the node.

```
[] compute(al) => outLst
   where proc :=
(Systole) dpu (assign(al,
  rAlu(1, reg(void,0,false), reg(void, 0, false),
          reg(x, 0, false), nop).
  rAlu(2, reg(kte, 3, true), reg(x, 0, false),
          reg(y, 0, false), *).
  rAlu(3, reg(kte, 3, true), reg(y, 0, true),
          reg(s3, 0, false), *).
  ...
```

Figure 18. a simple KressArray Description

The labeled rule Eval transforms the input list of rAlus taking the first one which is ready to process (both registers have the flag set to true), computing its output and them propagating it to other nodes.

```
rules for Dpu
//---------//
lstAlu, lstAluOut          : list[Alu];
al                         : AssgnLst;
aluRdy, newAlu             : Alu;
global
[Eval] dpu(lstAlu) => dpu(lstAluOut)
  where aluRdy :=()getAluRdy(lstAlu)
  if aluRdy != aluNull
  where newAlu := () exec(aluRdy)
  where lstAluOut :=
    () propagate(outReg(newAlu), update(newAlu, lstAlu))
  end
end
```

Figure 19. DPU rule in KressArray Description

One interesting aspect of using rewriting systems to model hardware is that it is relatively simple to provide rules that permit the symbolic processing of the inputs. In this case, providing numeric values to the system result in a numeric answer while providing symbolic values result in a symbolic result.

This aspect was tested against a simple KressArray with two nodes, computing the expressions:

x1 = x + dx;

c = a < x1;

Providing input (x, dx, a) = (3, 1, 10) results in (x1, c) = (4, 0). On the other hand, if one provides (x, dx, a) = (t, r, s) the result will be:

(x1, c) = ((t + r), s < (t + r))

This can be implemented by creating a partial ordering relationship between integers and variables. In this example, the type input was created as a supertype of integer and variable. The rules that compute the array values are duplicated, one set for integers and another set for variables. An example of those rules is presented bellow.

1. [] operate(i, j, *) => k where k := ()i * j end
2. [] operate(s, t, *) => (s*t) end

While rule 1 computes the integer value of i*j , the second rule returns the symbolic expression "(s*t)". This feature is very interesting for formal verification of a given mapping of C expressions into the systolic array.

For the sake of reconfiguration, the rAlu list is previously processed by a meta rule which specifies the operations that each node should perform. This is simply a list of operations in the form *.+.%.nil, for instance, which is assigned to each rAlu.

5. Conclusions

Our thesis is that we can profit from the discrimination between rewriting and logic to simplify the rewriting based specification, experimentation, simulation and verification of reconfigurable systems. By rewriting-logic even sophisticated dynamical reconfiguration appears a very natural mechanism to be simulated via logical strategies. Hardware description languages like VHDL and Verilog, and even SystemC, do not provide the degree of abstraction and flexibility found in rewriting systems. In fact, they do not compete in this field, since the detailed hardware design still must pass through a hardware description language (VHDL is the "assembly language" in this context). We do not need their architectural and circuit details for mapping an application onto a rDPA, nor design space exploration to optimize KressArray platforms [Na01].

Since digital systems get more and more complex, modeling the various architectural trade offs in the context of reconfigurable systems may benefit from the high abstraction level provided by rewriting-logic environments. Our experiments with ELAN targeted reconfigurable systolic arrays and the first results are encouraging. More sophisticated models are under development to study the possibilities of dynamical reconfiguration of systolic arrays.

6. References

[ArSh99] Arvind and X. Shen, *Using Term Rewriting Systems to Design and Verify Processors*, Technical Report 419, Laboratory for Computer Science - MIT, 1999. Also in IEEE Micro Special Issue on Modeling and Validation of Microprocessors, 1999.

[ARNe02] M. Ayala-Rincón, R. M. Neto, R.P. Jacobi, C. H. Llanos and R. W. Hartenstein, *Applying ELAN Strategies in Simulating Processors over Simple Architectures*. In B. Gramlich and S. Lucas Eds., Reduction Strategies in Rewriting and Programming, Elsevier ENTCS 70(6):20 pages, 2002.

[BaNi98] F. Baader and T. Nipkow, *Term Rewriting and all That,* Cambridge University Press, 1998.

[Bo98] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau and C. Ringeissen, *An overview of ELAN*, in Elsevier ENTCS, Vol. 15, 1998.

[BKKM02] P. Borovanský, C. Kirchner, H. Kirchner and P.-E. Moreau, *ELAN from a rewriting logic point of view*, pages 155-185 of [MOMe2002].

[CDELMOMQ02] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer and J. F. Quesada, *Maude: specification and programming in rewriting logic*, pages 187-243 of [MOMe2002].

[CiKi99] H. Cirstea and C. Kirchner, *Combining Higher-Order and First-Order Computation Using rho-Calculus: Towards a Semantics of ELAN*, Chapter 6 in Gabbay, D. M. and de Rijke, M. Eds., Frontiers of Combining Systems 2, Studies on Logic and Computation, 7, pages 95-121, Research Studies Press/Wiley, 1999.

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, *Introduction to Algorithms*, The MIT Press, 2001.

[DiFu02] R. Diaconescu and K. Futatsugi, *Logical foundations of CafeOBJ*, pages 289-318 of [MOMe02].

[Da89] M. Dauchet, *Simulation of Turning Machines by a Left-Linear Rewrite Rule.* 3rd Int. Conference on Rewriting Techniques and Applications RTA89, Vol. 355, pages 109-120 of *Lecture Notes in Computer Science*, 1989

[Da92] M. Dauchet, *Simulation of Turing Machines by a Regular Rewrite Rule.* Theoretical Computer Science, 103(2):409-420, 1992.

[HaKrRe95] R. Hartenstein, R. Kress and H. Reinig. *A Scalable, Parallel and Reconfigurable Datapath Architecture.* Sixth International Symposium on IC Technology, Systems and Applications - ISIC'95, Singapore, 1995. Available at www.kressarray.de.

[Ka00] D. Kapur. *Theorem Proving Support for Hardware Verification*, invited talk Third Int. Workshop on First-Order Theorem Proving, St. Andrews, Scotland, 2000.

[KaSu00] D. Kapur and M. Subramaniam. *Using and Induction Prover for Verifying Arithmetic Circuits.* Journal of Software Tools for Technology Transfer. 3(1):32-65, Springer Verlag, 2000.

[KaSu97] D. Kapur and M. Subramaniam, *Mechanizing Verification of Arithmetic Circuits: SRT Division.* In Proc. Seventeenth Conference on Foundations of Software Technology and Theoretical Computer Science. Vol. 1346 of LNCS, Springer-Verlag, 1997.

[KnBe70] D. E. Knuth and P. B. Bendix. *Computational Problems in Abstract Algebra*, chapter Simple Word Problems in Universal Algebras, pages 263-297. J. Leech, ed. Pergamon Press, Oxford, 1970.

[Kung87] S. Y. Kung. VLSI Array Processors. Prentice-Hall, 1987.

[MOMe02] N. Martí-Oliet and J. Meseguer, eds., *Special issue on Rewriting Logic and its Applications*, Theoretical Computer Science 285(2): 119-564, 2002.

[Me00] J. Meseguer. *Rewriting Logic and Maude: Concepts and Applications*, In L. Bachmair Ed., Eleventh Int. Conf. on Rewriting Techniques and Applications RTA 2000, LNCS, Vol. 1833, pages 1-26, Springer, 2000.

[Na00] U. Nageldinger et al. Kress Array Explorer: A New CAD Environment to Optimize Reconfigurable Datapath Array Architectures. ASP-DAC, Yodohama, Japan, Jan. 25-28, 2000.

[Na01] U. Nageldinger. Coarse-Grained Reconfigurable Architecture DesignSpace Exploration. Dissertation, Univ.Kaiserslautern, June 1, 2001.

[HHHN00] R. Hartenstein, M. Herz, T. Hoffmann, U. Nageldinger. *Kress Array Explorer: A New CAD Environment to Optimize Reconfigurable Datapath Array Architectures.* 5th Asia and South Pacific Design Automation Conference - ASP-DAC 2000, Yodohama, Japan, 2000. Available at www.kressarray.de.

[ShAr98a] X. Shen and Arvind, *Design and Verification of Speculative Processors*, Technical Report 400A, Laboratory for Computer Science - MIT, 1998. Also in Proc. of the Workshop on Formal Techniques for Hardware and Hardware-like Systems, Marstrand, Sweden, 1998.

[ShAr98b] X. Shen and Arvind, *Modeling and Verification of ISA Implementations*, Technical Report 400B, Laboratory for Computer Science - MIT, 1998. Also in Proc. of the Australasian Computer Architecture Conference, Perth, Australia, 1998.