Using Rewriting-Logic Notation for Funcional Verification in Data-Stream Based Reconfigurable Computing

Reiner W. Hartenstein[1], Ricardo P. Jacobi[2]
[1]Fachbereich Informatik
Kaiserslautern University of Technology
hartenst@rhhk.uni-kl.de
[2]Depto. De Ciência da Computação
Universidade de Brasília
rjacobi@cic.unb.br

Maurício Ayala-Rincón[3], Carlos H. Llanos[4]
[3]Departamento de Matemática
ayala@mat.unb.br
[4]Departamento de Engenharia Mecânica
llanos@unb.br
Universidade de Brasília
Brasília - Brazil

Abstract

Reconfigurable Systolic Arrays are a generalization of Systolic Arrays where node operations and interconnections can be redefined even at run time. This flexibility increases the range of systolic array's application, making the choice of the best systolic architecture to a given problem a critical task. In this work we investigate the specification and verification of such architectures using rewriting-logic, which provides a high level design framework for architectural exploration. In particular, we show how to use ELAN rewriting system to specify reconfigurable systems which can perform both arithmetic and symbolic computations.

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 circuit 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 [KnBe70], 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 with simple architectures [ShAr98a,ShAr98b,ArSh99], the rewrite-based description and synthesis of simple logical digital circuits [HoAr99] and the description of cache protocols over memory systems [RuShAr99,ArStSh01]. Also contributions in this field have shown how rewriting theory can be applied for the specification of processors (as Arvind's group does) as well as for the purely rewrite based simulation and analysis of the specified processors [ANJLH02]. To achieve this rewriting-logic has been applied, that extends the pure rewriting paradigm allowing for a logical control of the application of the rewriting rules by strategies [Me00,CiKi99]. Important programming environments based on the rewriting-logic paradigm are ELAN [CiKi99,BKKM02], Maude [Me00, Cla02] and Cafe-OBJ [DiFu02]. The impact of rewriting-logic as a successful programming paradigm in computer science as well as of the applicability of the related programming environments is witnessed by [MOMe02]. All our implementations and experiments were done in ELAN, since we consider of great flexibility its easy manipulation of strategies. However, for effects of model checking Maude has been proved to be more adequate.

Section 2 provides an introduction to basic concepts in rewriting theory and reconfigurable circuits. Section 3 presents the specification and simulation of reconfigurable systolic arrays. Section 4 discusses use of rewriting-logic for simulating reconfiguration and another approach for data driven systolic arrays and section 5 is the conclusion.

2. Background

An example of our typical target environments is mapping applications onto platforms like coarse-grained DPAs (DataPath Arrays) or rDPAs (reconfigurable DPAs). We assume, that such platforms are completely pre-debugged, so that only the related mapping source has to be verified. Using Term Rewriting Systems (s. section 2.1) in such an environment means to specify or verify designs from sources at abstraction levels being higher than that of languages like VHDL or Verilog. Such notations are much more compact and concise than with traditional hardware language source notations in EDA. An example is the input language of ELAN which is a parsable derivative of the math formula space.

This paper uses systolic arrays as demo examples (section 2.2). Systolic arrays, however, are special cases of super-systolic platforms like DPAs or rDPAs [HaKrRe95], which are data-stream-based [Ha03] pipe networks. (By the way: such platforms may also be emulated on larger FPGAs.) The only difference between systolic and super-systolic is the mapping method [Ha97]. Algebraic mapping or linear projection methods yield only solutions with linear uniform pipes which is restricted to the special case of applications with strictly regular data dependencies. But using simulated annealing, genetic algorithms or other optimization methods, permits any heterogeneous networks with free form pipes like zigzag, circular, or any much more wild shapes, and may also include forks and joins. The methodology introduced by this paper may be used for all kinds of data-stream-based hardwired or reconfigurable platforms.

2.1. Rewriting theory

We include the minimal needed notions on rewriting theory and rewriting-logic. For a detailed presentation of rewriting see [BaNi98].

A Term Rewriting System, TRS for short, is defined as a triple $\langle R, S, S_0 \rangle$ where S and R are respectively sets of *terms* and of *rewrite rules* of the form $l \rightarrow r$ ***if*** $p(l)$ being l and r terms and p a predicate and where S_0 is the subset of *initial terms* of S. l and r are called the left-hand and right-hand sides of the rule and p its condition.

In the architectural context of [ShAr98b], terms and rules represent states and state transitions, respectively.

A term s can be *rewritten* or *reduced* to the term t, denoted by $s \rightarrow t$, whenever there exists a subterm s' of s that can be transformed according to some rewrite rule into the term s'' such that replacing the occurrence of s' in s with s'' gives t. A term that cannot be rewritten is said to be in *normal* or *canonical form*. The relation over S given by the previous rewrite mechanism is called the *rewrite relation* of R and is denoted by $\rightarrow$. Its inverse is denoted by $\leftarrow$ and its reflexive-transitive closure by $\rightarrow^*$ and its equivalence closure by $\leftrightarrow^*$.

The important notions of *terminating* property (or Noetherianity) and Church-Rosser property or *confluence* are defined as usual. These notions correspond to the practical computational aspects as the determinism of processes and their finiteness.

- a TRS is said to be *terminating* if there are no infinite sequences of the form $s_0 \rightarrow s_1 \rightarrow \ldots$

- a TRS is said to be *confluent* if for all *divergence* of the form $s \rightarrow^* t_1$, $s \rightarrow^* t_2$ there exists a term u such that $t_1 \rightarrow^* u$ and $t_2 \rightarrow^* u$.

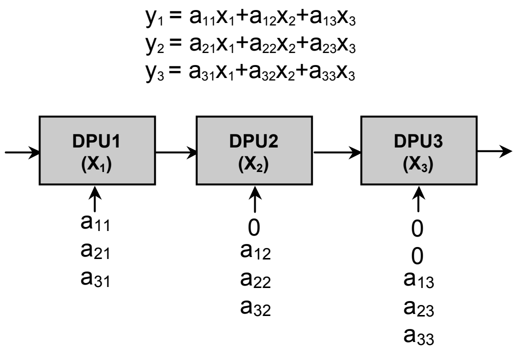
The use of the subset of initial terms S_0, representing possible initial states in the architectural context (which is not standard in rewriting theory), is simply to define what is a "legal" state according to the set of rewrite rules R; i.e., t is a legal term (or state) whenever there exists an initial state s [illegible] S_0 such that $s \rightarrow^* t$.

Using these notions of rewriting one can model the operational semantics of algebraic operators and functions. Although in the pure rewriting context rules are applied in a truly non deterministic manner in the practice it is necessary to have a control of the ordering in which rules are applied. Thus rewriting theory jointly with logic, that is known as rewriting-logic, has been showed of practical applicability in this context of specification of processors since they may be adapted for representing in the necessary detail many hardware elements involved in processors. Moreover, other important fields of hardware design such as verification and synthesis of logical circuits may be benefited from the simplicity and versatility of this theoretical framework.

2.2. Systolic arrays and reconfigurable systems

A systolic array is a mesh-connected pipe network of DPUs (datapath units), using only nearest neighbor (NN) interconnect [Ku78, Ku87]. DPU functional units operate synchronously, processing streams of data that traverse the network. Systolic arrays provide a large amount of parallelism and are well adapted to a restrict set of computational problems, i.e., those which can be efficiently mapped to a regular network of operators.

Figure 1 shows a simple systolic example of a matrix-vector multiplication. The vector elements are stored in the cells and are multiplied by the matrix elements that are shifted bottom-up. On the first cycle, the first cell (DPU1) computes $x_1{*}a_{11}$, while the second and third cells (DPU2 and DPU3) multiply their values by 0. On the second cycle, the first cell computes $x_1{*}a_{21}$, while the second cell computes $x_1{*}a_{11} + x_2{*}a_{12}$, where the first term is taken from the first cell and added to the product produced in second cell. In the third cycle, the third cell produces the first result: $y_1 = x_1{*}a_{11} + x_2{*}a_{12} + x_3{*}a_{13}$. In the following two cycles y_2 and y_3 will be output by the third cell.



$y_1 = a_{11}x_1+a_{12}x_2+a_{13}x_3$
$y_2 = a_{21}x_1+a_{22}x_2+a_{23}x_3$
$y_3 = a_{31}x_1+a_{32}x_2+a_{33}x_3$
DPU1 (X_1)
DPU2 (X_2)
DPU3 (X_3)
a_{11} a_{21} a_{31}
0 a_{12} a_{22} a_{32}
0 0 a_{13} a_{23} a_{33}

Figure 1: Vector – matrix computation.

There are several alternative configurations of functional cells, each one tailored to a particular class of computing problems. However, one of the main critics to systolic arrays is its restriction to applications with strictly regular data dependencies, as well as its lack of flexibility. Once designed, it is suitable to support only one particular application problem.

The limitations of systolic arrays may be circumvented by using reconfigurable circuits, the most representative of them being the FPGAs (Field Programmable Gate Arrays). An FPGA can have its behavior redefined in such a way that it can implement completely different digital systems on the same chip. Fine grain FPGAs may redefine a circuit at the gate level, working with bit wide operators. This kind of architecture provides a lot of flexibility, but takes more time to reconfigure than coarse grain reconfigurable platforms (rDPAs: reconfigurable DPAs). These work with word wide operators that are slightly less flexible but more area efficient and take much less time to reconfigure than the fine grain ones. The design of reconfigurable systolic architectures [HaKrRe95, HHHN00] aims to overcome the restriction of pure systolic circuits while keeping the benefits of a large degree of parallelism. In the reconfigurable approach, the operations performed by each functional unit as well as the interconnection among them may be reconfigured in order to be adapted to different applications. Moreover, it is possible to change the configuration of the circuit during run time, an approach called dynamic reconfiguration, which broadens even more the architectural alternatives to implement applications in hardware.

3. Specification and Simulation of Systolic Arrays via Rewriting-Logic

Here we show how rewriting-logic can be applied to specify simple systolic arrays for vector and matrix multiplication using the ELAN system. In these systems we can consider as the reconfigurable part the constants in each component (DPU as in Figure 1), here called *MAC* (Multiplier/Adder).

Vector multiplication: ELAN provides a very flexible programming environment where the user defines the syntax and semantics of data types (called *sorts*) and operations to be used in the program. Figure 2 shows the syntax of the data types in the ELAN program that models the vector multiplication. In the left side of a definition the data is specified using a combination of text and place holders which are represented by an '@' character. For instance, an element of sort *Port* is defined as *port(@, @)*. The sort of the parameters as well as the sort of the element itself is defined in the right side of the definition. In this example, the two parameters of *port* are an integer and a Boolean, and the resulting element *port(int, bool)* is of sort *Port*. A *MAC* data is composed of six elements. The sort of the elements is, respectively, *int*, for the identifier, two of sort *Port* and two of sort *Reg* for the ports and registers and one of sort *Const* for the respective constant component of the multiplier vector. The processor sort *Proc* consists of four components: three of sort *MAC* and one of sort *DataStream*. The *DataStream* is described as an object with three components of sort *list[Data].*

```
operators global
 @                          : ( int ) Const;
 port(@,@)                  : ( int bool ) Port;
 reg(@,@)                   : ( int bool ) Reg;
 '[' @,@,@,@,@,@ ']'        : ( int Port Port Reg Reg Const ) MAC;
 '<' @ @ @ @ '>'            : ( MAC MAC MAC DataStream ) Proc;
 ( @ @ @ )                  : ( list[Data] list[Data] list[Data] ) DataStream;
 @                          : ( int ) Data;
end
```

Figure 2. Sorts of the MAC in ELAN

The rule named *sole*, used to describe the behavior of the processor during each cycle of the execution is given in figure 4. Informally, the rule is *fired* when the expression being processed matches the left side of the rule. It is replaced by the expression produced at the right side of the rule, which is again matched against the set of rules that define the program. Rules can be named for future reference when defining strategies. The rule name *sole* appears between square brackets in the beginning of the rule. Observe that after one-step of reduction applying this rule all necessary changes in the specified processor are executed. First, notice that the data *d1*, *d2* and *d3* at the top of the *DataStream*, are removed from the three lists of data and placed in the first ports of the three *MACs*.

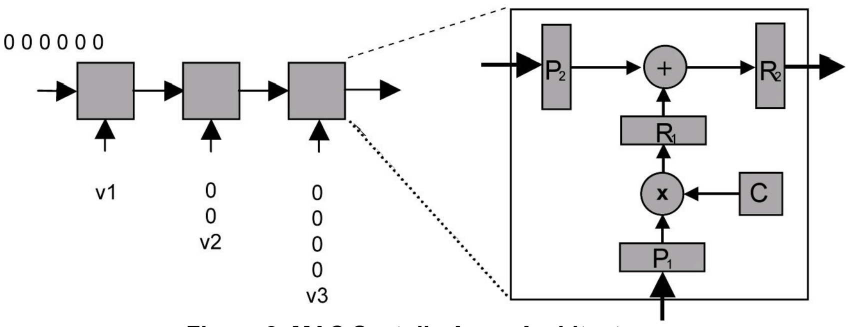


Figure 3. MAC Systolic Array Architecture

The multiplications between the contents of each first port vpi_1 and the corresponding constant c_i are placed in the first register of each *MAC*, for i=1,2 3 and 3 and the additions between the first register vri_1 and the second port vpi_2 are placed in the second port of each MAC, for i=1,2 and 3. Zeros preceding each operator v_i are included to synchronize the two operations executed in each *MAC*. Finally, observe the data transfer from the second register vri_2 of each *MAC* to the second port of the next component $vp_{(i+1)2}$ for i=1 and 2. All that is simultaneously done by only one application of the rewriting rule *sole*.

With respect to the timing aspects of this example, the model assumes a clocked operation like traditional systolic architectures. There is no handshake between nodes and each application of rule *sole* corresponds to a single clock cycle. Thus, each node in fig. 3 takes two clock cycles to produce its output. The synchronization between nodes for the first values is achieved introducing pairs of zero values, as illustrated in figure 3, and the Boolean flags used to synchronize nodes could be omitted.

Executing all steps with a singe rewriting rule could appear artificial in other contexts of computer science, such as semantics of programming languages and in general theory of computing. Nice but relatively complex theoretical results can be related with the possibility of having a unique rewriting rule which simulates a (universal) Turing machine [Da1989, Da1992]. This nontrivial theoretical development may be better understood when we relate a sole rule with the execution of a "cycle" of a processor.

```
rules for Proc
  d1,d2,d3 : int;                                    // variables for input data
  l1,l2,l3 : list[Data];                             // lists of input data
  vp11, vp12, vp21, vp22, vp31, vp32  : int;         // ports
  vr11, vr12, vr21, vr22, vr31, vr32  : int;         // registers
  c1, c2, c3 : int;                                  // constants
global
  [sole]
<[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(d1,true),port(0,true),reg(vp11*c1,true), reg(0+vr11,true),c1 ]
  [2, port(d2,true),port(vr12,true),reg(vp21*c2,true), reg(vp22+vr21,true),c2 ]
  [3, port(d3,true),port(vr22,true),reg(vp31*c3,true), reg(vp32+vr31,true),c3 ]
  (l1 l2 l3)  >
 end
end
```

Figure 4. ELAN Description of the *Sole* Rule.

For our example we will consider as simple mechanism of reconfiguration the possibility of changing the constants in each *MAC*. Then a computation with our systolic array consists of two stages: a reconfiguration stage, where the constants are set and the subsequent processor execution with the previously defined rule *sole*.

Figure 5 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, because the resulting expression will match against the left side of the rule again and again. Thus, for controlling its application, we define a logical strategy, called *withconf.*

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). This normalization is an available ELAN strategy, which applies the rewriting rules given as argument (in our case the rule sole) until a normal form is reached.

```
 [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 5. *conf* Rule for Reconfiguration

Matrix Multiplication: figure 6 shows the matriz multiplication structure and the description of its components is given in figure 7. Using the previous approach (that is, specifying a sole rule) implies the use of an excessive number of variables, which is not directly supported in ELAN. In fact, we would 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 would need 16 variables for describing the two 4x data streams. This could be done by enlarging the ELAN capacity 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, to simulate the internal process into each MAC component and the propagation of data between each component to their North and East connected MACs.

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 a consequence of the form in which data should be 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.

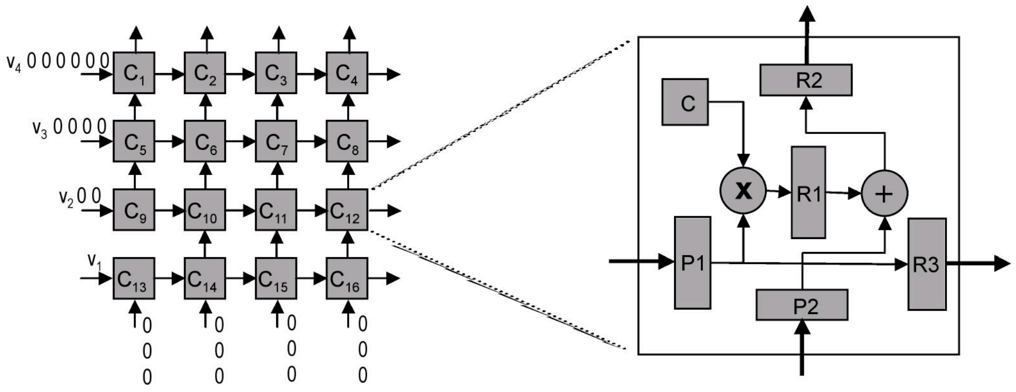


Figure 6. Systolic Matrix-vector multiplication

```
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 7. A 4✕4 Systolic array Description

All these rules are very similar and a selected group of them is presented in the Figure 8. The rules for the South (*mac01*, *mac02*, *mac03*, *mac04*) and West (*mac01*, *mac05*, *mac09, mac13*) *MACs*, called boundary components of the processor, load the data (*dS* and *dW*) in the head of the corresponding list of the data stream (*IS1, IS2, IS3, IS4* and *IW1, IW2, IW3* and *IW4*). Moreover, 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*. Consequently, different orderings for applying these rules completing a whole cycle of the processor are possible. For instance, we could take the ordering: *mac16, mac12, mac08, mac04, mac15, mac11, mac07, mac03, mac14, mac13, mac10, mac09, mac06, mac05, mac02, mac01.*

In the Figure 9 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(onecycle)*. In this rewriting-logical environment, our specification could be easily modified allowing the interpretation of parts of the processors as reconfigurable components.

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 matrix F_4 of the Fast Discrete Fourier Transform (FFT).

```
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 8. A selected set of rules for matrix-vector multiplipy.

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

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

Figure 9. onecycle **strategy for rule application.**

Figure 10: strategy working over processor.

The last is specified by a strategy additional, that is presented at the Figure 11, which before to the simulation of the normalization executes the rewrite rule F_4. F_4 transforms any given state of the processor into another where the reconfigurable 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 11. 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 directly extended to different kind of *MACs*, where other components are considered to be reconfigurable.

```
[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 11: rule for FFT Transformer

4. Alternative Models

4.1 Variable Size Systolic Arrays

One limitation of the ELAN models presented above is that the rules are defined for a specific systolic architecture. 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.

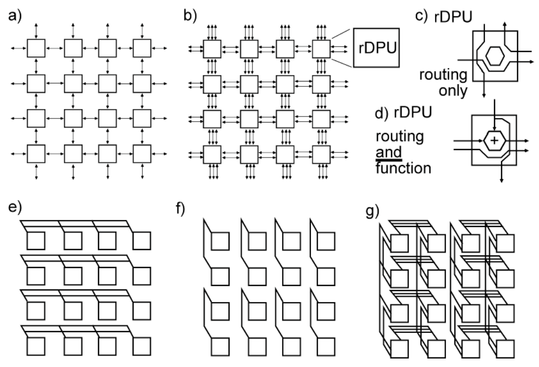


Figure 12. KressArray family design space: (a, b) NN fabrics examples; (e, f, g) backbus (BB) fabrics examples; rDPU configuration: c) routing only configuration, d) routing and function configuration

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. 12 c and d). Figure 12 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 13 shows a detailed example of NN ports featuring individual path width and individual mode (in, out, or bi-directional).

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 implements a dataflow model of computation. 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) second level interconnect fabrics with resources like buses and bus segments (for family member examples see fig. 12 e, f, g).

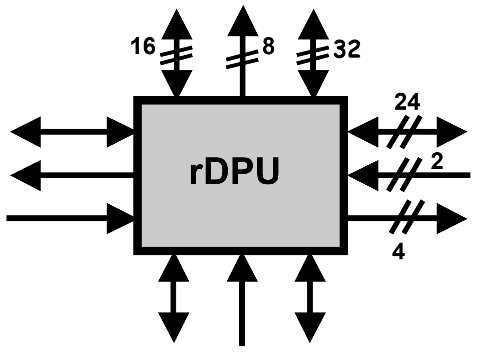


Figure 13: a KressArray family member example illustrating individual port mode and path width

The DPU nodes of both, systolic arrays and rDPAs, may operate in a clocked mode, or asynchronously, where each operation 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. To allow the specification of variable size systolic arrays, the nodes are stored on a list of arbitrary length. The designer provides the interconnections among nodes through signals, implemented with variables that connect 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. The operations of the nodes are dynamically specified along with the data that is to be processed by the array. To illustrate this kind of modeling lets consider a practical example: a KressArray that computes the differential equation loop body, given in figure 14.

This array is defined by a list of 12 nodes, numbered from left to right and top to bottom. The first 4 nodes are presented in figure 15. Each *rDPU* node 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, specified in the form "x=5.y=2…". Then, rule *dpu()* is applied to the array produced by *assign()*. The *Systole* keyword is a strategy that normalizes the rule taking the first result (basic ELAN strategy *first one*) produced by *dpu()* rule, presented in Figure 16.

$x_1 = x + dx;$
$u_1 = u - (3*x)*(u*dx) - 3*y*dx;$
$y_1 = y + u*dx;$
$x = x_1;$
$u = u_1;$
$y = y_1;$
$t_1 = x < a;$

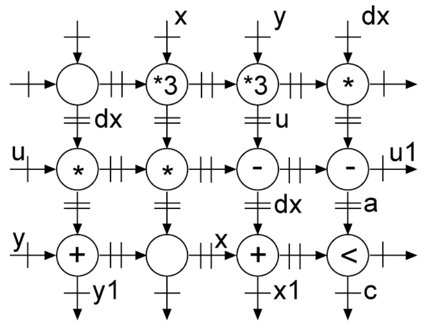


Figure 14. KressArray implementation of the differential equation.

Label *Eval* names the rule to allow calling it from the strategies. *dpu()* rule produces a call to itself taking as parameter the systolic array where a *ready* node was processed. 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.

```
[] 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(s2, 0, false),  *).
                          rAlu(3, reg(kte, 3, true), reg(y, 0, true), reg(s3, 0, false),  *).
                          rAlu(4, reg(s3, 0, false), reg(y, 0, false), reg(s4, 0, false),  *).
  …
```

Figure 15. A simple KressArray Description

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. The labeled rule *Eval* transforms the input list of rAlu's 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 16. DPU rule in KressArray Description

The circuit can be simulated providing a query in the form "compute(x=2.y=4.dx=5.u=2.a=10.null) end", for instance, to ELAN. The result, for a single pass of the algorithm, is returned in the form "u1=41.y1=14.x1=7.c=1.null".

The reconfiguration is implemented by providing a list of operators that should be assigned to each node. This is easily implemented by extending the *compute()* rule to include a list of operations as another parameter. Figure 17 exemplifies this new rule including a configuration list as parameter and the call of a new rule *assignCFG()* to apply this configuration to the KressArray. The parameter *cfglst* is in the form: "*.+.nop. … . nil". Thus, the expression that describes the KressArray is first processed by the reconfiguration rule followed by the input variable assignments and finally simulated through the *dpu()* rule.

4.2 Symbolic Computation

One interesting aspect in using rewriting systems to model hardware is that it is relatively simple to define rules that permit the symbolic processing of the inputs. In this case, the designer can provide a set of variables as inputs and the system produces a set of equations in terms of those variables as the output.

```
[] compute(al, cfglst) => outLst
    where proc :=
(Systole) dpu (assign(al, assignCFG(cfglst, 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(s2, 0, false),  nop).
                                            rAlu(3, reg(kte, 3, true), reg(y, 0, true), reg(s3, 0, false),  nop).
  …
```

Figure 17. Including reconfiguration in KressArray.

In the previous example, consider the set of equations:

```
x1 = x + dx;
c = a < x1;
```

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

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

The symbolic expressions obtained this way can be used to formally verify the algorithm implemented in the systolic array, by checking them against the specification extracted from the C language description. The formal verification is one of the subjects for future research.

Implementing symbolic processing in ELAN can done by creating a partial ordering relationship between integers and variables. In this example, the sort *input* was created as a supertype of *integer* and *variable*, as illustrated bellow.

```
…
sort Reg Op Dpu Pair input variable; end     // sort declaration
operators global
  @              : (int) input;               // this is a type casting, specifying that both int and variable
  @              : (variable) input;          // sorts should be considered as input sort as well
```

Given this type casting, then the rules may be defined taking into account any combination of parameter sorts. For example, one operation may be defined only for integer values, or it may be defined for integer and variable, and so on. In this example, the operations should be defined in terms of the supertype input:

```
@ * @            : (input input) input;       // operations defined over input sort operands
@ + @            : (input input) input;       // and producing input sort result
@ / @            : (input input) input;
```

The two expressions above are implemented by two nodes of the KressArray. An extract of the ELAN definition of this simple example is presented below.

```
reg(@,@)          : (input bool) Reg;                  // register with value and ready flag.
rAlu(@,@,@,@)     : (Reg Reg Reg Op) Node;             // reconfigurable alu.
dpu(@, @)         : (Node Node) rDpu;                  // a simple datapath unit.
```

The dpu consists of two nodes, each one containing three registers and a operation. Each register holds an input sort data and the ready flag. The transformations rules are applied in a similar fashion, as shown in figure 18. Each rule is triggered by the availability of the operands, which is specified by the ready flag. Those rules basically change the registers status. The input registers are marked as processed and the output registers receive the results, which may be a numeric value or a symbolic expression. The rAlu() rule compute the result and returns it as a register value. Note that there is the rAlu() data type, which has four components and the rAlu() rule, used for computing, which has three components. The rAlu() rule is defined as follow:

```
[] rAlu(reg(d0, b0), reg(d1, b1), op) => reg(d, b)
        where d := () operate(d0, d1, op)
        where b := () true
                end
```

The matching engine of ELAN will select the appropriate operation based on the sort of the operands d0 and d1. Rules (1) and (2) below are defined either over integer parameters *i* and *j* or over variable sort parameters *s* and *t*.

```
(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)", in terms of the actual values of *s* and *t*.

```
[] dpu(rAlu(a1, b1, r1, op1), rAlu(a2, b2, r2, op2)) => dpu(rAlu(na1, nb1, nr1, op1), rAlu(na2, b2, r2, op2))
        if ready(a1) and ready(b1)
        where nr1  := ()rAlu(a1, b1, op1)
        where na1 := ()used(a1)
        where nb1 := ()used(b1)
        where na2 := ()nr1
end
[] dpu(rAlu(a1, b1, r1, op1), rAlu(a2, b2, r2, op2)) => dpu(rAlu(a1, b1, r1, op1), rAlu(na2, nb2, nr2, op2))
        if ready(a2) and ready(b2)
        where nr2  := ()rAlu(a2, b2, op2)
        where na2 := ()used(a2)
        where nb2 := ()used(b2)
end
```

Figure 18. Rules for symbolic computation example.

5. Conclusions

In this paper we have shown how rewriting-logic can be used to model and simulate reconfigurable systolic arrays. The examples presented in this paper illustrate the specification and simulation of both synchronous and dataflow models of such systems. The rewriting-logic environment ELAN used in this work permits a high-level functional style description of systems. The discrimination between rewriting and logical strategies in those systems is used to simplify the purely rewrite based specification, experimentation, simulation and verification of reconfigurable systems. By rewriting-logic even sophisticated dynamic 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. Moreover, with a relatively small effort it is possible to generate symbolic expressions from the model, which helps the formal verification of the selected architecture. Currently, more sophisticated models are under development to study the possibilities of dynamic 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.

[ANJLH02] 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. Borovansk_, C. Kirchner, H. Kirchner, P.-E. Moreau and C. Ringeissen. *An overview of ELAN*, in Elsevier ENTCS, Vol. 15, 1998.

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

[Cla02] 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

[Ha97] R. Hartenstein (invited paper). *The Microprocessor is no more General Purpose: why Future Reconfigurable Platforms will win;* Proc. of IEEE International Conference on Innovative Systems in Silicon, (ISIS`97), Austin, Texas, USA, 1997

[Ha03] R. Hartenstein (keynote). *Data-stream-based Computing and Morphware*; joint 33rd Speedup and 19th PARS Workshop, Basel, Switzerland, 2003

[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, Yokohama, Japan, 2000. Available at www.kressarray.de.

[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.

[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.

[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.

[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.

[Ku78] H.T. Kung, C. E. Leiserson. *Systolic Arrays for VLSI*; Sparse Matrix Proc. 1978, Society for Industrial and Applied Mathematics, 1979, pages 256-282.

[Ku87] 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.

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

[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.