FINITE STATE MACHINES : COMPOSITION, VERIFICATION, MINIMIZATION : A CASE STUDY

PAUL AMBLARD[1], FABIENNE LAGNIER[2], MICHEL LEVY[3]
1: TIMA-CMP, 46 av. Félix Viallet, 38031 GRENOBLE Cedex
2: Vérimag, Centre Equation, 2 avenue de Vignate, 38610 GIERES
3: LSR-IMAG, B.P. 72, 38042 St MARTIN D'HERES Cedex
Université Joseph Fourier, Grenoble, France.
(Paul.Amblard, Fabienne.Lagnier, Michel.Levy)@imag.fr

Abstract— **A deep understanding of circuit behaviour is a prerequisite for any validation process (simulation, formal verification, test generation). We propose to use a tool which gives complete and optimized representations of sequential circuits allowing the designer to understand the accurate behaviour of the circuit. A detailed example is introduced to help reader's understanding. For obvious reasons, we choose a small size circuit. The example comes from our experience ([2]) in computer architecture and digital design education.**

Keywords—**Finite State Machines, sequential circuits, data-path control part, FSM minimization.**

INTRODUCTION

Before to design sequential synchronous circuits, the designers have often to deal with their representations as Finite State Machines (FSM). A Finite State Machine (or Finite Automaton) can be described in several different ways :

- The FSM is described by a list of inputs (Σ), states (Q) and outputs (Ω) completed by the description of the transition ($\delta : \Sigma \times Q \rightarrow Q$) and output ($\lambda$) functions. We shall name this description the *specification* of the FSM.

Generally the transition function δ is described by a kind of table, giving the new state as a function of the inputs and the previous state. Representations based on bubbles and arrows are also used. It is well known [10] that this description is not unique : two FSMs can be equivalent. We also know that a minimal form exists.

- The FSM is described by a list of gates and flip-flops with their interconnections. We shall name this detailed description of the implementation at the logic level the *logic implementation* of the FSM.

Here we consider only synchronous implementation, assuming the existence of a CLOCK and of a RESET signal. The description is not unique : the designer (or the CAD tool) can choose different binary codes for the informations (inputs, states, outputs) and this will give different implementations.

- We must note that a FSM circuit can be described as a combination of several sub-circuits. In the frame of this paper we restrict ourselves to sub-circuits sharing the same CLOCK signal. Different combinations exist : serial combination (inputs of automaton 2 are the outputs of automaton 1), parallel combinations (possibly some of the inputs of automaton 2 are a part of outputs of automaton 1 and some inputs of automaton 1 are outputs of automaton 2). This idea is already presented in the conclusion of 1956 Moore's paper ([12]) : *Certain inputs of each machine are connected to the outputs of others, and other inputs and outputs of the individual machines are used as the inputs and outputs of the composite machine.* (p 153)

- In many situations the circuit is described as a set of registers, multiplexors, arithmetic operators and busses. This type of description, often considered as being at the *Register Transfer Level* (RTL) allows to deal with complex automata with a simple description. The commands of MUXes, ALUs are the inputs, reports, such as flags are the outputs. Data inputs and outputs play another role. A simple circuit with two 8 bits registers, an Arithmetic and Logic Unit, an input from the external world and a bunch of commands can have 65 536 states ($2^{(8+8)}$) and a transition function where each state can have any of the 65 536 successors. A description at the "table" level would be impractical. Such RTL descriptions can be based on either boolean or integer type.

An example of description by bubbles-and-arrows or in RTL style is given in figure 1.

The work presented here consists in composing automata and exploring the result of the composition. This, as will be shown, allows the designer to deeply understand the synchronisation aspects in the circuit. FSM minimization then allows to deal with abstraction in preparing the formal verification of the circuit.

The basics of our approach are very simple : given a circuit description of a FSM *logic implementation* or a boolean based RTL description, a tool first computes the *specification* expansion of the FSM and then delivers the minimal FSM equivalent to the given one.

Obviously this is the reverse task compared to the very common synthesis tools available in all standard CAD packages. So we shall have to explain the interest of such a task. This is done in a first section. A second section shows the language and environment used for our work. A last section contains a detailed example.

THE GOALS AND PRINCIPLES

Let us recall briefly what is composition of automata and what kind of exploration we want to do with this composition.

What is *composition of automata* ?

At an abstract level, things are quite simple. Let

- A= $(Q_a, \Sigma_a, \Omega_a, \delta_a, \lambda_a, q_a^0)$ an automaton,

Q_a is the set of states, Σ_a the set of inputs, Ω_a the set of outputs, δ_a the transition function, λ_a the output function and q_a^0 the initial state,

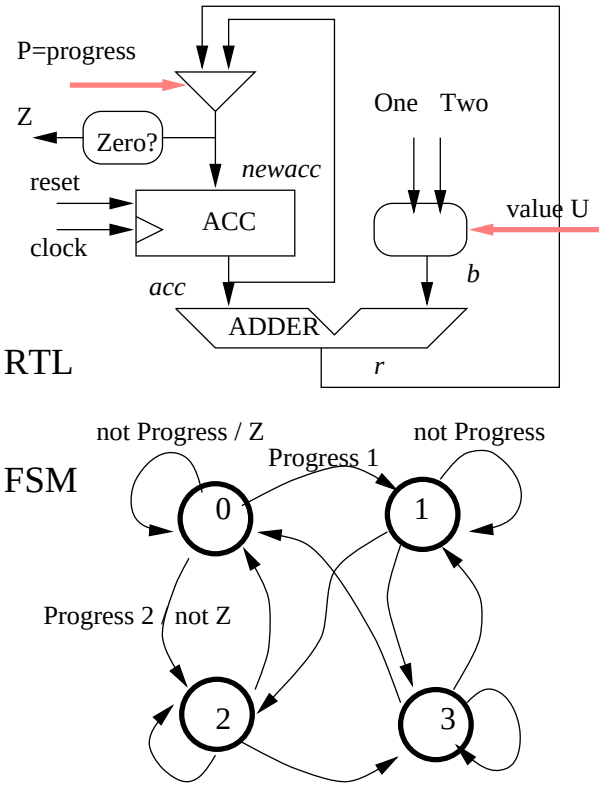

Fig. 1. Structure of the data path and state diagram of the automaton for a 2 bits size. The numbering of the states corresponds to the values in the Accumulator. In the automaton, "Progress 1" means "progress" and add the value 1. Output Z is true for 3 combinations : state 0 and no Progress, state 3 and Progress 1 and state 2 and Progress 2.

- B= $(Q_b, \Sigma_b, \Omega_b, \delta_b, \lambda_b, q_b^0)$ an automaton,

The composition C=$(Q, \Sigma, \Omega, \delta, \lambda, q^0)$ is defined as such :

- $Q = Q_a \times Q_b$
- $q^0 = (q_a^0, q_b^0)$
- Σ is given ;

we define $f : \Sigma \times \Omega_b \longrightarrow \Sigma_a$

- we define $g : \Sigma \times \Omega_a \longrightarrow \Sigma_b$
- Ω is given ; we define $h : \Omega_a \times \Omega_b \longrightarrow \Omega$.

Composition of two Moore automata is different from composition of a Moore automaton and a Mealy automaton :

A : Moore, B : Moore

$$\begin{aligned} \delta((p_a, p_b), e) &= (\delta_a(p_a, f(e, \lambda_b(p_b))), \\ &\quad \delta_b(p_b, g(e, \lambda_a(p_a)))) \\ \lambda((p_a, p_b)) &= h(\lambda_a(p_a), \lambda_b(p_b)) \end{aligned}$$

A : Moore, B : Mealy

$$\begin{aligned} \delta((p_a, p_b), e) &= (\delta_a(p_a, f(e, \lambda_b(p_b, g(e, \lambda_a(p_a))))), \\ &\quad \delta_b(p_b, g(e, \lambda_a(p_a)))) \\ \lambda((p_a, p_b), e) &= h(\lambda_a(p_a), \lambda_b((p_b), g(e, \lambda_a(p_a))) \end{aligned}$$

Let us notice that Mealy-Mealy composition could be unsafe, introducing combinational loops. Shiple & al. propose a method to solve the problem [13].

What is *exploration* ?

Our technique is based on exploration of the circuit behaviour. Exploration means that a designer is not already completely certain about *What must be done ?* It is too early to give any kind of implementation, formal specification or any similar description. Exploration is what you do on draft paper, with a pencil. You are *entering* into your design and you need to dig into it. You try, in fact, to understand what your circuit will be (or *would be* ?). Exploration is certainly not for a full circuit but for a part of circuit, for a mechanism, for a hardware trick. As an example we could be interested in the good synchronization between a control part and a data-path part. Composing Moore-Moore, or Moore-Mealy types does not give equivalent results from the temporal point of view.

We shall then explore the result of the composition of the two automata.

The tools used in this phase help you to get all the informations from a rough description. They make a kind of Computer Aided Draft. But your draft paper deals with formal calculations or proofs when needed. In a further section, an example will be presented to make clear this approach.

Comparison of *exploration* with other approaches

Let us compare the principles of this technique to other approaches.

• A first characteristic of our approach is its relation to simulation.

- In a first step, simulation allows the designer to check consistency between the implementation and the intention. This part is known to be difficult and unsafe. The behaviour of the implementation is seen by timing diagrams and the informations obtained are dependent upon the testbench prepared. The problem of elaborating a good testbench is highly difficult. Exploration can give some complete informations while simulation cannot. In exploration, we do not need to give a testbench. We have, in a certain way, ALL the possible testbenches. Obviously it can be too much. But the behaviour obtained by this technique is complete.

- In a second step, simulation can take into account some informations extracted from layout steps. Our approach cannot replace these computations managing the wires and gates delay. Exploration is not useful at this level.

• A second characteristic is the relation to model checking. We use a formal approach, and associated tools. The LUSTRE compiler is in fact a model checker. It contains the functionalities to build and minimize an automaton. Generally, the people involved in model checking use equivalence relations on the states to avoid combinatorial explosion ([5]). For example Herbrand Automata have been introduced ([6]) to model the same kind of composition of automata. The authors use them to verify some parts of complex micro-processors. One of the authors of these Herbrand Automata, A. Pnueli, explains the difference between *exploration* and *model checking* (of software) in the foreword of [5] : *While simulation and testing explore some of the possible behaviors and scenarios of the system, [..], formal verification conducts an exhaustive exploration of all possible behaviors.* Obviously testing of hardware has other goals.

In our approach these complete Finite State Machine structures *are the* useful information.

```
node mux1bit (i, t, e: bool) returns (s:bool);
let
  s = (i and t) or (not i and e);
tel;

node add1bit (a,b, ret_in :bool)
     returns (som, ret_out: bool);
let
  som = a xor b xor ret_in ;
  ret_out = a and b or a and ret_in or
            b and ret_in;
tel;

node addNbits (const N: int; a, b: bool^N)
     returns (r: bool^N);
var carry : bool^(N+1) ;
let
  carry[0] = false ;
  (r[0..(N-1)], carry[1..N]) =
  add1bit (
      a[0..(N-1)],
      b[0..(N-1)],
      carry[0..(N-1)]);
tel;

node B_mealy_gene (const N: int; p,u: bool)
     returns (z: bool);
var
b,r : bool^N ; -- adder inputs outputs
newacc , acc : bool^N; -- inp_out accu
cumuland : bool^(N+1); -- "and" test zero

let
   b[0] = u; -- plus 1
   b[1] = not u; -- plus 2
-- extension to N bits
   b[2..(N-1)] = false^(N-2);

   r =  addNbits (N , acc, b);
   newacc = mux1bit(p^N, r, acc) ;
   cumuland [N] = true;
   cumuland[0..(N-1)] = not newacc[0..(N-1)]
                        and
                        cumuland[1..N] ;
   z = cumuland [0];

   acc[0..(N-1)] = edgeDflip
                   (newacc[0..(N-1)]);
tel;

const width = 3;
node B_mealy_3_i (p,u: bool) returns (z: bool);
let
   z =  B_mealy_gene (width, p,u);
tel;
```

Fig. 2. Lustre description of a data-path

A work based on "Parallel composition of FSM" ([17]) allows to solve some kind of equations in automata. Common authors investigate a kind of "decomposition of automata" ([7]).

THE LANGUAGE AND TOOLS

In this section, we give some details on the techniques used to describe the source FSMs and about the results given by the tools we use.

How do we describe ?

All the descriptions are given in the language LUSTRE ([8] and [9]). LUSTRE is close to Lola, the language used by N. Wirth in his book. ([16])
Description may be of different types :

- Circuits described as a set of nodes : the nodes contain logic gates and edge-triggered D-type flip-flops. The only data type is boolean. (Cf node `mux1bit` or `add1bit` figure 2)

Let us notice that these description can take into account different codes of the states.

- Circuits described as a hierarchical or compositional set of nodes. The nodes can be different (cooperating) automata. The language is such that, basically, all the automata share the same clock. Due to this feature, LUSTRE is often refered to as a *synchronous* language.
- Generic circuits of size N, dealing with boolean vectors of size N. Registers have N flip-flops. Adder can be N bits wide. (Cf node `addNbits` hereafter). Notice the "implicit" repetition of `add1bit` in `addNbits`. N must be instanciated before effective use. (Cf node `B_mealy_3_i`) This allows us to have a same description for any N-bits circuits, we only need to change one constant. The same feature exists in VHDL.

The example in figure 2 is given for illustration, it corresponds to the circuit drawn in figure 1. It will be explained in a further section.

What do we obtain ?

We shall *compile* the circuit description. The compiler delivers a description of the automaton given in input. The description is based on the set of states and the two functions : transition function and output function. If the input description contained several automata, the compiler computes the product automaton. The description of the result automaton is given either in an internal textual form, or in C language, ready for compilation, or in a graphical form. It could as well be given in VHDL or an other Hardware Description Language. The execution of the C version gives the same results than a simulator.
Another tool allows to minimize this automaton. It gives the minimal automaton equivalent to the proposed one.
What uses can be done with the result of this composition-expansion-minimization ?

- A first use is to check the circuit obtained by a classic synthesis tool : all the common commercial CAD tools contain a package implementing a Finite State Machine by flip-flops and gates from a description of the automaton by states and transitions. Our tool contains, in a certain form, the reverse tool.
- A second very important use is allowed by this tool, but we shall not enter into the details in the frame of this paper : if we describe two logic implementation of a same FSM, based on different codes of the states, and if we add a comparator on the outputs, we can check the equivalence of the two FSM (Let us notice that the comparator is virtual in this case). They are equivalent if the comparator delivers always "True". This is computed by the compiler. We use this approach in education [3].
- In this paper we deal with a third use, exploration, based on the careful manual analysis of the result of the expansion process. For instance we shall study the synchronization aspects of a composition and the signification of a formal minimization.

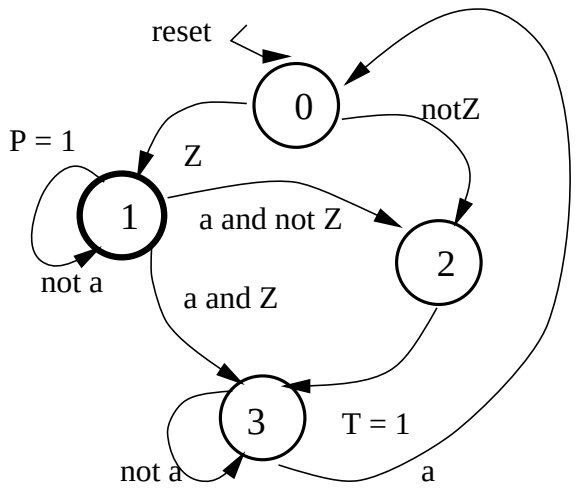

Fig. 3. The state diagram of the controller. The inputs are the two wires a and Z. The output T is true only in state 3, the output P is true only in state 1.

THE EXAMPLE

We present in this section an example composed of a controller and a data-path part.
To make the paper as self-consistent as possible, we made drastic simplifications and limited ourselves to a very simple example.

Control Part - Data-Path Part composition

Our example is a circuit composed of two parts : a data-path and a controller. This decomposition is well known and allows description at different levels. This kind of specification can be done by a variety of Algorithmic State Machines. These descriptions for synthesis are available in VHDL or Verilog. Combinational blocks of the data-path and timing in the controller are chosen. The description of the circuit can be done as two Finite State Machines. For controllers a table approach is generally admitted (or bubbles and arrows). The outputs are the commands towards the data-path. Data-pathes are described at the RTL level. The designers prefer to speak about a 10 bits accumulator and an adder then about an automaton with thousands of states. We shall conform to this technique to *describe* the data-path, but we shall explore the structure of the equivalent *specification* automaton.

The circuit data-path

The data-path is a simple adder + accumulator structure. (Cf. figure 1) It has two commands :
The first command **progress** or **P** allows the addition. When unactive, the accumulator remains unchanged.
The second command **value** or **U** selects a value to add : we made our choice between adding 1 or 2. Any other couples of values could have been chosen as well.
The circuit delivers a boolean output **Z** true iff the new value of the accumulator is Zero. (The automaton will be of Mealy type)
The circuit is described as a generic N bits wide thing. The text presented previously is for N = 3. A restriction

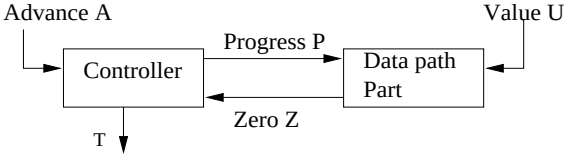

Fig. 4. Composition of the controller and the data-path.

to N = 2 gives the automaton in figure 1. For N = 3 we would obviously have 8 states instead of 4. There is a combinatorial explosion in the number of states. We are aware of this fact.

The circuit controller

Our control part is a simple 4-state machine. It is described by figure 3. The input **a** comes from external world. In states 1 and 3, **a** allows **a**dvance of the controller. In states 1 and 3, if **a** is zero, the controller remains in the same state. The output **T** is a simple **t**est. The other output **P** will be the **P**rogression command towards the accumulator. The input **Z** is boolean (it will be the output of the data-path).

Composition

We linked these two parts according to the scheme given in figure 4 and asked the Lustre compiler to compute the automaton equivalent to the product automaton so obtained. The compiler first gives the full automaton, then it is possible to minimize it. Let us examine (figure 5) and comment the two versions of product automata : minimized or not minimized.
In both description we are invited to consider macro-states named PC=0, . . . , PC=3. They correspond to one state of the controller. PC=0 means that the controller is in state 0. For one state of the controller, there can be different values possibles in the data-path. They appear on the diagram. A ”state” of the data-path is the current value in the accumulator. It appears as Acc=1 or Acc=2.
Figure 7 gives the complete product of these two automata as a set of couples. The composition first shows that synchronisation between the two parts is correct. Specifically, the value of Z tested by the controller is the value computed in the data-path at the previous clock cycle.
Another information is obtained : only 15 among the 16 possible states can be obtained. Careful analysis of automaton suggests that it is impossible to have a state where Control part could be in state PC=2 while data-path would be in a state with Acc=0.
Another property can also be expressed : ”Once the accumulator is different of 0 in the controller state (PC=2), it will become definitively impossible to change the accumulator.

Interpretation of minimization

An understanding of the meaning of minimization is given for this example.
- in the non-optimized version, the values of the accumulator is always meaningfull, whatever the state of the

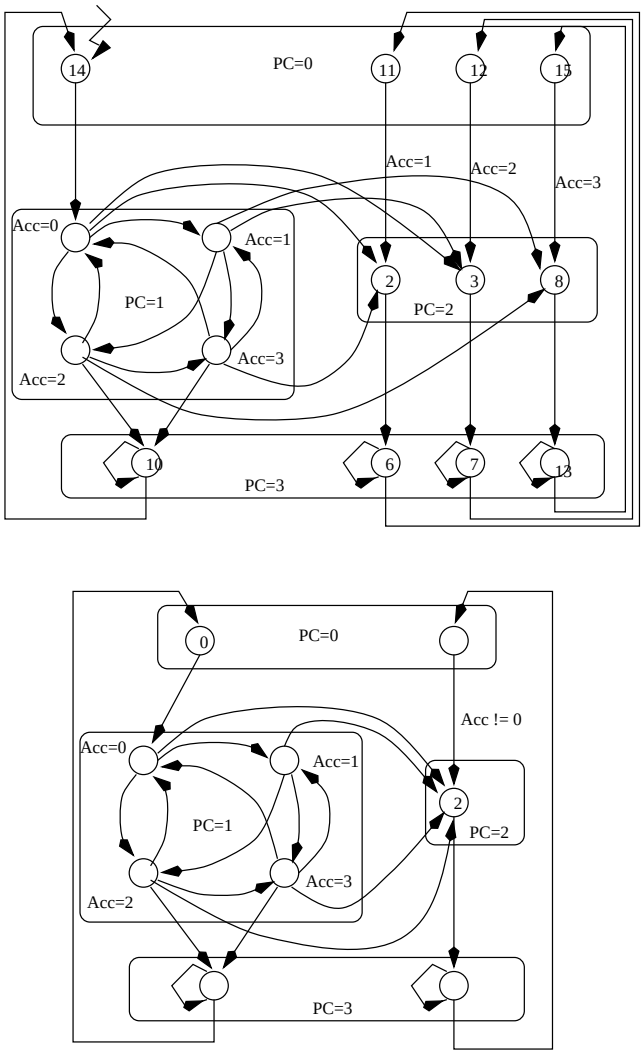

Fig. 5. Two versions of the composition : on the top the original version, on the bottom the minimized version. PC is the state of the Control Part, Acc is the state of the data-path part. Output T is true only in state PC=3.

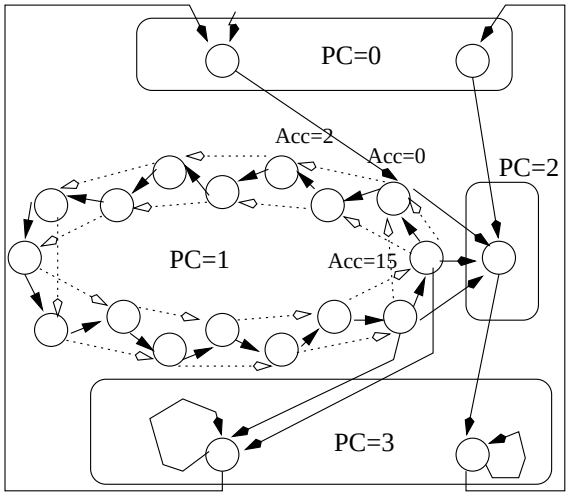

Fig. 6. Minimized composition of the controller and a 4 bits wide data-path. All the arrows from all the states of the ring towards state where PC=2 are not drawn. The different types of arrows in the ring correspond to the plus 1 and plus 2 arithmetic operations.

	-a-u	-au	a-u	au
0,0 ”14”	0,1 ”1”	0,1 ”1”	0,1 ”1”	0,1 ”1”
1,0 ”1”	1,2 ”5”	1,1 ”4”	2,2 ”3”	2,1 ”2”
2,1 ”2”	3,1 ”6”	3,1 ”6”	3,1 ”6”	3,1 ”6”
2,2 ”3”	3,2 ”7”	3,2 ”7”	3,2 ”7”	3,2 ”7”
1,1 ”4”	1,3 ”9”	1,2 ”5”	2,3 ”8”	2,2 ”3”
1,2 ”5”	1,0 ”1”	1,3 ”9”	3,0 ”10”	2,3 ”8”
3,1 ”6”	3,1 ”6”	3,1 ”6”	0,1 ”11”	0,1 ”11”
3,2 ”7”	3,2 ”7”	3,2 ”7”	0,2 ”12”	0,2 ”12”
2,3 ”8”	3,3 ”13”	3,3 ”13”	3,3 ”13”	3,3 ”13”
1,3 ”9”	1,1 ”4”	1,0 ”1”	2,1 ”2”	3,0 ”10”
3,0 ”10”	3,0 ”10”	3,0 ”10”	0,0 ”14”	0,0 ”14”
0,1 ”11”	2,1 ”2”	2,1 ”2”	2,1 ”2”	2,1 ”2”
0,2 ”12”	2,2 ”3”	2,2 ”3”	2,2 ”3”	2,2 ”3”
3,3 ”13”	3,3 ”13”	3,3 ”13”	0,3 ”15”	0,3 ”15”
0,3 ”15”	2,3 ”8”	2,3 ”8”	2,3 ”8”	2,3 ”8”
2,0 ”..”	3,0 ”10”	3,0 ”10”	3,0 ”10”	3,0 ”10”

Fig. 7. Complete composition of the automata. In couples, the first component is the state of Control Part, the second one is the state of the Data-Path Part, (the value stored in the accumulator). For each state the number, as in figure 5, is given between quotes. The four next states, for the four combinations of boolean values a and u are given. State (2,0) has no number, it does not exist.

controller. The reader will however notice that the value cannot change while the automaton has entered in one of the states 11, 12, 15, 2, 3, 8, 6, 7, 13. (top right of figure 5) This is normal, progression of Accumulator is only possible in state PC=1 of the controller. And when PC=2, Accumulator does not change, and Z cannot be modified. It is so impossible to go back to state PC=1.

- in the minimized version, this value is usefull only in state PC=1. This information is usefull because the controller forces **Progress** in the accumulator. The accumulator will progress of 1 or 2, depending on a condition unknown from the controller. The result of the progression is necessary to establish the next state of the controller, according to the value of **Z** (Accumulator = 0).

The same kind of structure can easily be obtained for more than 2 bits wide data-path. We generated the automata, minimized or not, for 2, 3, and 4 bits and we easily checked the regularity in the structure. Obviously this circuit is strongly sensible to arithmetic facts. Figure 6 shows the combination of a 4 bits wide data-path and the

controller. Techniques such as McMillan's *uninterpreted functions* [11] could not be applied. *" A common reason for the "non-essential" growth of the state space is the presence of a significant data path component in a circuit. [ ] Thus, it is often possible to abstract the large set of values 2^n in an n-bit data path to a small set for the purposes of analysis. (Note that this is not directly feasible for arithmetic circuits) " [15]*

CONCLUSION

The kind of exploration presented in this paper is based on human understanding of computer generated automata. This approach is a complement of simulation or formal validation. It is very fruitfull because *exploration* gives, in a certain way, all the behaviors of a circuit, while simulation gives only the ones chosen by the benchmak's designer. Let us remain aware that this approach is also very difficult.

Our other experiments, not presented here because they were too big, show that 100 states is a maximal complexity. It means that we must simplify drastically a mechanism to study it. For instance, the size of the automaton obtain in figure 6, with only 21 states is discouraging. The non-optimized version had 64 states. We did not draw it in this paper. For another study ([4]) we obtained a 6500 states automaton and it was impossible to manage it by hand.

The main goal of this technique is to study a mechanism, in this present paper synchronization between control and data-path parts. Our example shows also some arithmetic regularities. But our technique is not *aware* of this fact. Other techniques have to be used to take this kind of aspects into account ([1], [5],...).

Acknowledgements

The authors express thanks to their colleagues working in Lustre environment. They benefit also from discussions with othen teachers in computer architecture.

References

[1] Arvind and X. Shen, Using Term Rewriting Systems to Design and Verify Processors, IEEE Micro, May-June 1999, pp 36-46.

[2] P. Amblard, J.C. Fernandez, F. Lagnier, F. Maraninchi, P. Sicard et P. Waille, *Architectures Logicielles et Matérielles*, Dunod, 2000.

[3] P. Amblard, F. Lagnier and M. Levy, Introducing Digital Circuits Design and Verification Concurrently, Proceedings of the 3rd European Workshop on Microelectronics Education, Aix en Provence, 18-19 May 2000, Kluwer, pp 261-264.

[4] P. Amblard, A Finite State Description of the Earliest Logical Computer : the Jevons' Machine, Mixed Design of Integrated Circuits and Systems, (Ed A. Napieralski, Z. Ciota, A. Martinez, G. De Mey, J. Cabestany), Kluwer, 1998.

[5] E. Clarke, O. Grumberg, D. Peled, *Model Checking*, The M.I.T. Press, 1999.

[6] W. Damm, A. Pnueli, S. Ruah, Herbrand Automata for Hardware Verification, Proceedings of CONCUR98, 9th International Conference, Nice, France, September 1998, pp 65-83.

[7] M.D. Di Benedetto, A. Sangiovanni-Vincentelli, T. Villa, Model Matching for Finite-State Machines, IEEE Transactions on Automatic Control, Vol 46, No 11, November 2001, pp 1726-1743

[8] N. Halbwachs, P. Caspi, P. Raymond and D. Pilaud : The Synchronous Data-flow Programming Language Lustre, Proceedings of the IEEE, pp 1305-1320, September 1991.

[9] N. Halbwachs, F. Lagnier and C. Ratel : Programming and Verifying Real-time Systems by Means of the Synchronous Data-flow Programming Language Lustre, IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems, September 1992, pp 785-793.

[10] J. Hopcroft, J. Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, 1979.

[11] K. McMillan, Verification of Infinite State Systems by Compositional Model Checking, CHARME 1999, Bad Herrenhalb, september 1999, pp 219-233. (LNCS 1703)

[12] E. Moore, Gedanken-experiments on Sequential Machines, Automata studies (Edited by C. E. Shannon and J. McCarthy), Princeton University Press 1956, pp 129-153.

[13] T.R. Shiple, G. Berry, H. Touati, Constructive analysis of cyclic circuits. EDTC, Paris, March 1996.

[14] M. Srivas, H. Rueβ and D. Cyrluk, Hardware Verification using PVS, Formal Hardware Verification, Ed T. Kropf, 1997, pp 156-205. (LNCS 1287)

[15] P.A. Subrahmanyam, Towards Verifying Large(r) Systems : A strategy and an experiment. CHARME 1993, Arles, May 1993, pp 135-154. (LNCS 683)

[16] N. Wirth : *Digital Circuit Design*, Springer-Verlag, 1995.

[17] N. Yevtushenko, T. Villa, R. Brayton, A. Petrenko, A. Sangiovanni-Vincentelli, Solution of Parallel Language Equations for Logic Synthesis, International Conference on Computer Aided Design, November 2001, pp 103-110.