UNIVERSITÄT PASSAU
Fakultät für Mathematik und Informatik

Verification of Systolic Arrays in M2L(Str)

Tiziana Margaria

Fakultät für Mathematik und Informatik
Universität Passau
Innstr. 33
D-94032 Passau (Germany)
tel: +49 851 509.3096
fax: +49 851 509.3092
tiziana@fmi.uni-passau.de

MIP-9613
Juli 1996

Abstract

VLSI designs often show regular structures, where issues like temporal and spatial recursivity, and bidirectionality play a central role. This document introduces the modelling of a class of regular VLSI circuits, namely iterative systolic arrays, in a particular second-order logic, and presents fully automatic verification of pipeline properties for an example from the literature.

Contents

1 Introduction **1**

2 Related Work **1**

3 Iterative Linear Systolic Arrays **3**

4 The Specification Language **5**

5 The Verification Environment **8**

6 Case Study: the Example of [RhSo93] Revisited **9**

6.1 The Original Statement of the Problem 9
6.2 The Gate-level Library 11
6.3 Problem 1: The Basic Cells 13
6.4 Verification 1: Behaviour of the Single Cells 16
6.5 Verification 1A: The Re-implemented Basic Cells 18
6.6 Problem 2: The Iterative Linear Arrays 19
6.7 Verification 2: Behavioural Equivalence of the Arrays 20
6.8 Verification of Abstract Properties 21
6.9 Library-based Verification 22

7 Lessons Learned **23**

7.1 The Language 23
7.2 Modelling Hardware in this Logic 23
7.3 Evaluation of the Tool 24
7.4 Understanding of this Case Study 25

8 Conclusions **26**

1 Introduction

Formal verification of hardware circuits requires both a reliable formal description system and an adequate verification tool. This paper proposes a new method for the automatic verification of a class of systolic systems, based on the use of monadic second order logic as a modelling language (the formal description system) and Mona as the corresponding verification tool.

We concentrate here on the description and verification of a class of **systolic arrays**, which is intensively used e.g. in the implementation of *hardware accelerators*[1] and of encoders/decoders for security purposes. Systolic systems exhibit a regular behaviour both over *time* and over *structure*. Traditionally, this behaviour has been formalized for verification purposes by means of recursion, and it has led to induction-based proofs. Unfortunately neither recursive modelling nor induction-based reasoning are familiar to hardware designers, which accounts in large part for the scarce penetration achieved so far by formal methods in the common practice of circuit design. The modelling language proposed here is much closer to the modelling practice of hardware designers since it avoids recursion and, as usual in Hardware Description Languages, it uses parameterization to capture the meta-structure contained in a systolic array. In particular,

- Parametrization over *time* allows the description of the sequential behavior of a single processor in the form of difference equations.
- Parametrization over *structure* is particularly suited for VLSI implementations where the same basic cell is often instanced many times to yield a regular structure.

Together with the fully automated tool support, this yields modular definition of circuits and hierarchical verification, as required in industrial practice.

After a discussion of related work in Sect. 2, Sect. 3 introduces the Iterative Linear Systolic Arrays, Sect. 4 presents the chosen specification language, and Sect. 5 describes the verification environment. The long Sect. 6 discusses in detail the application of the method to the modelling and verification of a case study. Finally, Sect. 7 summarizes the lessons learned so far, and Sect. 8 draws our conclusions.

2 Related Work

Traditional approaches to parametric hardware verification with formal techniques resort to general purpose methods based on inductive reasoning within either first or higher-order logic. First order logic based theorem provers such as Boyer's and Moore's NQTHM [Pier89, Hunt89, Pier94], or the Larch Prover [Alle94] have been used for years to handle parametric designs, as well as several higher order logic systems like HOL [Gord86, CaGM87, Melh88], VERITAS [HaDa86], or NUPRL [Cons86, BaDV89],

[1] I.e., devices capable of performing the same functions of, for example, typical Abstract Data Type objects, like e.g. the priority queue of [CaMP89].

and there are studies which compare both approaches [AnCD91]. In all of them, verification becomes an activity bound to interactive, computer assisted theorem proving. Thus the user must employ tactics (as in the HOL system) or suggest appropriate lemmas. While such approaches are very general and powerful, their use is mainly restricted to verification experts, as opposed to engineers and CAD designers. Also the approaches of [Wolp86, SiGe87, BrCG89, KuMi89, WoLo89] for model-checking, language containment, and process verification require interaction.

We know of two other approaches to fully automate the verification of parametric circuits. Both are based on symbolic methods:

- Rho and Somenzi propose methods to automatically derive invariants from the structure of iterative sequential circuits by resorting to automata-based methods and to BDD-based symbolic manipulation of automata [RhSo92, RhSo93]. The derivation process constructs the invariant by greatest fixpoint iteration over state-minimal deterministic representations of the corresponding regular languages. It is successful whenever it stabilizes up to isomorphism. As no characteristic criterion for stabilization is given, the precise class of circuits that can be handled is unclear.
- Gupta and Fisher [GuFi93] introduce a canonical representation for inductive boolean functions (IBF) which resembles an inductive extension of BDDs covering certain classes of inductively-defined hardware circuits. Verification is carried out by symbolic tautology checking on the IBF representation of the circuits. Their results are the most similar to ours, as they cover essentially the same class of circuits, which they call *linear* parameterized sequential circuits. However, besides being complicated, the required coding in a special inductive format has the following drawbacks:
  - The coding is bound to a very fine-grained analysis of the recursion pattern and does not directly reflect the structure of the original description. Thus diagnostic information would require to re-establish the connection to the original description.[2]
  - The treatment of parametric outputs is very tricky and leads to complicated representations of hierarchical structures.

In contrast, our approach allows a quite straightforward structure-preserving representation of the argument circuits and therefore good support for error diagnosis. Moreover, as will be illustrated in the examples, no additional effort is required to treat parametric outputs. This is achieved by using as formal description language the Monadic Second-Order Logic on Strings (M2L(Str)), which Alonzo Church proposed already 30 years ago as an appropriate formalism for reasoning about bitvectors [Chur63]. In fact, this logic is decidable. However, it is decidable only in non-elementary time: the worst-case complexity is a stack of exponentials of height proportional to the size of the formula, which seems a good

[2] Perhaps this is the reason for not being considered in the paper.

reason for it having been considered impractical so long. Fortunately, relevant problems are usually far better behaved and can be solved automatically in reasonable time[3]. The parametric hardware examples treated so far required CPU times ranging from fractions of a second to a few minutes.

3 Iterative Linear Systolic Arrays

Systolic arrays [SuMe77, KuLe79] are synchronous networks of parallel processors. They show nice properties from both an engineering and a mathematical point of view, by combining multiprocessing and pipelining techniques together with the more theoretical concepts of cellular automata and algorithms.

A systolic array is built of several identical processors, each performing an elementary computation every clock tick, since all operands reach the proper processor simultaneously. The overall computation is therefore carried out in real parallelism and no waiting is necessary. To ensure proper communications, systolic arrays do not allow free broadcasting: within a time unit every processor can only exchange data with its nearest neighbours or with the host. Since systolic systems are special-purpose machines, they form a global computing system together with the host. Because of their simple and regular architecture, these systems also show a regular structure with simple connections between adjacent processors. This structural property is immediately translatable into high performance and low overhead of support components, since it implies simple and regular geometries for the layout.

In this paper we concentrate on *linear*, i.e., one-dimensional, systolic arrays, also called *Iterative* Sequential Systems by Somenzi and Rho. The structure of such a system, called *iterative network*, is illustrated in Fig. 1: the array is constructed as a linear chain of interconnected instances C_1 to C_n of the same *basic cell* C. Each cell C_i communicates with the environment via *local inputs* y_i and x_i^1 to x_i^p (which are the primary inputs and outputs of the systolic array and which are externally controllable). Additionally, there are uni- or bidirectional *communication channels* between adjacent cells. The inputs coming from the extreme left and right model the communication with the host, and are called *boundary conditions*.

An iterative network is unilateral if the communication channels carry information only in one direction, otherwise it is bilateral. An iterative network is circular if its cells are connected in a ring. *Sequential* (resp. combinational) networks have sequential (resp. combinational) circuits as basic cells. Sequential iterative networks correspond to two-dimensional combinational iterative networks. An *iterative system* is the class of all iterative networks having the same basic cell and boundary conditions, and a different (finite) number of cells.

In the following, we deal with the most general case, namely **sequential iterative systems** of any kind (uni-, bidirectional or circular). The problems we are going to attack involve therefore parameterization over time, due to the sequential behaviour of

[3]The so popular BDD encodings are also in general exponential, but large classes of hardware circuits have manageable polynomial representations.

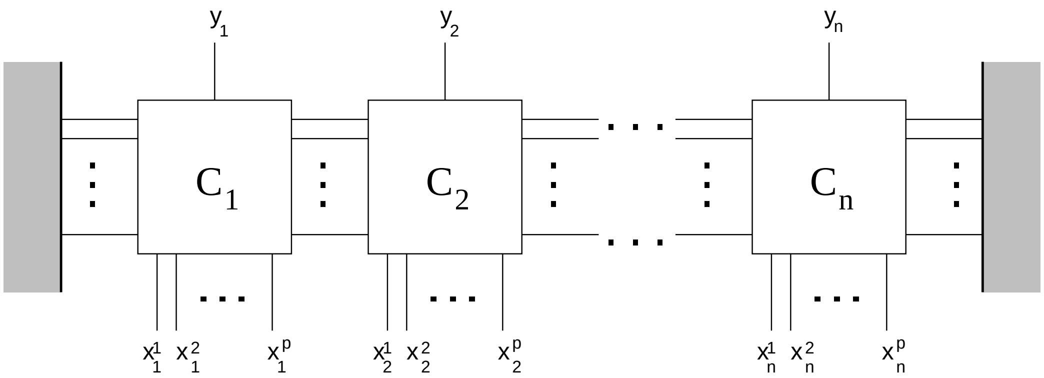


Figure 1: Structure of Iterative Linear Systolic Arrays

the basic cells, and parameterization over structure, since we consider arrays with a finite, but arbitrary number of cells. The internal structure of the basic cell of such systems is usually specified as a Finite State Machine and implemented in hardware at the gate-level. In particular, at this granularity we will regard each cell as a "slice" of the overall array.

Finite-State Machines

In systolic arrays all processors have exactly the same structure: this leads to a modular structure of the overall system. Each processor is built of a certain number of finite-state machines and of a connection mesh. The machines belong to the class of the *Moore machines*, while the connection mesh takes cares of both internal and external communications, i.e., within a single processor or between different processors and with the host, respectively.

Moore machines, also called *state-output* Finite State Machines, show the desirable property that outputs depend on the state, but not on the inputs. Another class of FSMs, the *Mealy machines* or *transition-output* FSMs, output a combinational function of both the state and the current inputs.

Moore Machines.
According to the formalization of [Koha70], a Moore machine is defined as a quintuple:

$$
\begin{aligned}
Moore \;&\equiv\; (\, Q,\, I,\, O,\, \delta,\, \lambda\,) \\
Q \;&\equiv\; \text{finite set of } \textit{internal states} \\
I \;&\equiv\; \text{finite set of } \textit{input symbols} \\
O \;&\equiv\; \text{finite set of } \textit{output symbols} \\
\delta \;&\equiv\; Q \otimes I \rightarrow Q, \\
\lambda \;&\equiv\; Q \otimes I \rightarrow O,
\end{aligned}
$$

where δ and λ denote the combinational *state transition function* and *ouput transition function*, respectively.

In this mathematical model, time is an integer parameter which is in charge of counting

the number of clock cycles or state changes.
The state $q(t+1)$ and output $o(t+1)$ at time $t+1$ are given by

$$\begin{aligned} q(t+1) &\equiv \delta\,(q(t),\ i(t)) \\ o(t+1) &\equiv \lambda\,(q(t),\ i(t)) \end{aligned}$$

Mealy Machines
In the same way a Mealy machine is also a quintuple:

$$Mealy \equiv (\ Q,\ I,\ O,\ \delta,\ \lambda\ ),$$

but the next state $q(t+1)$ and the current output $o(t)$ are in the form

$$\begin{aligned} q(t+1) &\equiv \delta\,(q(t),\ i(t)) \\ o(t) &\equiv \lambda\,(q(t),\ i(t)) \end{aligned}$$

Semisystolic systems include Mealy machines in their processors. This model shows how input signal changes can reach the outputs within the same clock cycle, i.e., signals are allowed to *ripple* through the combinational logic of different machines independently from synchronous timing constraints. The need of settling of combinational logic can force to resort to long clock periods, whose duration increases with growing system size. The usual additional constraint that outputs cannot be connected to form a cycle in the communication graph prevents the resulting feedback causing unclocked oscillations.

Systolic systems only allow Moore machines, thus avoiding all rippling problems and allowing designers to take the *number* of clock ticks as a discrete measure of time, since the clock cycle duration is not related to the system's size. As we only deal with systolic arrays, the fact that they only contain nearest-neighbour interprocessor communication makes the longest propagational delay negligible with respect to clock duration, so that the chosen level of abstraction in measuring time is legal.

Even if an efficient implementation of regular circuits resorts to purely systolic design, it is often easier to include Mealy machines which take care of the global communication needs. Transformations between the two classes are possible, by appropriately retiming portions of the basic cell. Structural transformations from a semi-systolic to a totally systolic architecture and viceversa concern in fact retimimg of regular arrays, which can be performed e.g. according to the principles found in [Leis82]. In the following we will therefore consider purely systolic systems, implemented resorting to Moore machines only.

4 The Specification Language

The *monadic second-order logic on strings* (M2L(Str)) is one of the most expressive decidable logics known, and it precisley captures regular languages (see [Thom90]). For the syntax we choose to follow the input syntax of the Mona tool (see Sect. 5). The basic operators are reported in Figure 2, where we distinguish logic terms denoting positions `t`, string expressions `T`, and formulas `F`. To understand the modelling of iterative systolic

```
t ::=  0  |  $  |  p  |  t  +o  i | 0 + i
T ::=  all |  P  |  compl(T)  |  T1  inter  T2 |  T + i
F ::=  true  |  t1  =  t2  |  t1 < t2  |  T1  =  T2  |  t in T  |
      ~F  |  F1 & F2  |  All p: F  |  All P: F
```

Figure 2: A basic syntax for M2L(Str)

arrays, we need to be familiar with the interpretation of the logic over both *time* and *structure*.

M2L(Str) for waweforms

The interpretation of the logic over waveforms is needed to model and reason about the sequential circuit implementing the basic cells. Here,

- First-order terms `t` describe *discrete time points* or *clock cycles*. `0` is the start cycle and `$` the final cycle of the considered time interval. The operators `+` and `+o` denote (with slight differences) addition modulo the interval length, where `i` ranges over natural numbers. Finally, `p` ranges over first-order variables.
- Second-order terms `T` denote Boolean *signals* over the considered time interval, represented as the set of cycle times in which the signal is set to 1. `all` is the constant 1 signal, `inter` is the pointwise 'and' of two signals, `compl(T)` denotes the pointwise complement of `T`, and `+` is the operator which shifts `T` right by `i` steps, thus corresponding to the i-clock delay operator on signals. Finally, `P` ranges over signals.
- Formulas `F` specify the behaviour of a circuit over all finite observation intervals. The atomic formulas are equations `t1 = t2` and inequations `t1 < t2` of clock cycles, equations on signals `T1 = T2`, and the construct `t in T`, which is true if signal `T` has value 1 in cycle `t`. Negation `~` and conjunction `&` are as usual. Finally, we can quantify over cycles and signals, `All p: F`, `All P: F`.

M2L(Str) for bit-sliced structures

Here first-order terms are taken to represent bit-slice indices, and second-order terms represent bit-vectors of generic length. The formula `t in T` then means "the bit with index `t` is set to 1 in vector `T`." Interpreted in the domain of iterative systolic systems, single bit-slices correspond to single cells, while a bit-vector ranges over the whole iterative system. In this setting,

- First-order expressions `t` describe positions of cells in an array. Since the operator `+o` which denotes position shift modulo the string length, the logic is adequate for modeling circular systems too.
- Second-order expressions `T` range over entire linear arrays.

Derived M2L(Str) operators

Many of the connectives given here, like e. g. position variables and their connectives, are only included for convenience since they may be encoded within the logic using second-order variables (see e.g. [Thom90]). Similarly, dual connectives like, e.g.,`false`, the empty string `empty`, and `union`, implication `=>`, equivalence `<=>`, existential quantification `Ex` on strings are available, as well as a short form for Boolean (propositional) variables, represented as `@p`, over which quantification is possible too. Predicate definitions are equalities terminated by semicolons, and comments are introduced by the symbol `#`.

Semantic models are constructed by converting formulas to automata as sketched in [HJJK95]. For any formula `F` that is not a tautology, a minimal length counterexample can be extracted from the corresponding automaton. This feature is exploited for fault detection, diagnosis and testing (see [Marg96, MaMe96]).

Example: A D-type flip-flop

The following predicate `dff` corresponds to a D-type flip flop, with input `D` and outputs `Q` and `Qn`, where `Qn` is the complementary output to `Q`:

```
dff(D,Q,Qn) = ((All t: t<$ => ((t +o 1 in Q <=> t in D) &
                                (t +o 1 in Qn <=> t notin D)))
                  & 0 notin Q & 0 in Qn);
```

This flip flop, which will be used as storage element (state holding device) in the case study of the next section, is modelled using parametrization over time. The second order variables `D`, `Q` and `Qn` model the values observed on the input/output ports along a discrete time axis. Their positions represent in fact observation (or sampling) points, with one observation point for each clock cycle.

Observation starts at time $t = 0$, where the initialization conditions (last line) say that signal `Q` is low (position `0` is not contained in the set `Q`, written in the logic as `0 notin Q`), and its complement `Qn` is high (position `0` is contained in the set `Qn`, written in the logic as `0 in Qn`).

The behaviour of the D flip-flop at observation points $t > 0$ is the following:

> for every finite prefix of the (infinite) waveforms observable at the ports (expressed by the precondition `t<$`), position `t +o 1` is set at output `Q` iff position `t` is set at input `D`.

This means that changes on the signal `D` are followed by the same change on the signal `Q` with a delay of one clock cycle, which is exactly the behavioural definition of the D flip-flop element. Similarly, the same changes on signal `D` cause a change to the opposite logic value on signal `Qn` with the delay of one clock cycle. That is, signals `Q` and `Qn` have always the opposite logic values, as it is expected for complementary outputs.

The behaviour of `Qn` could be equivalently defined for every clock cycle by using this property as

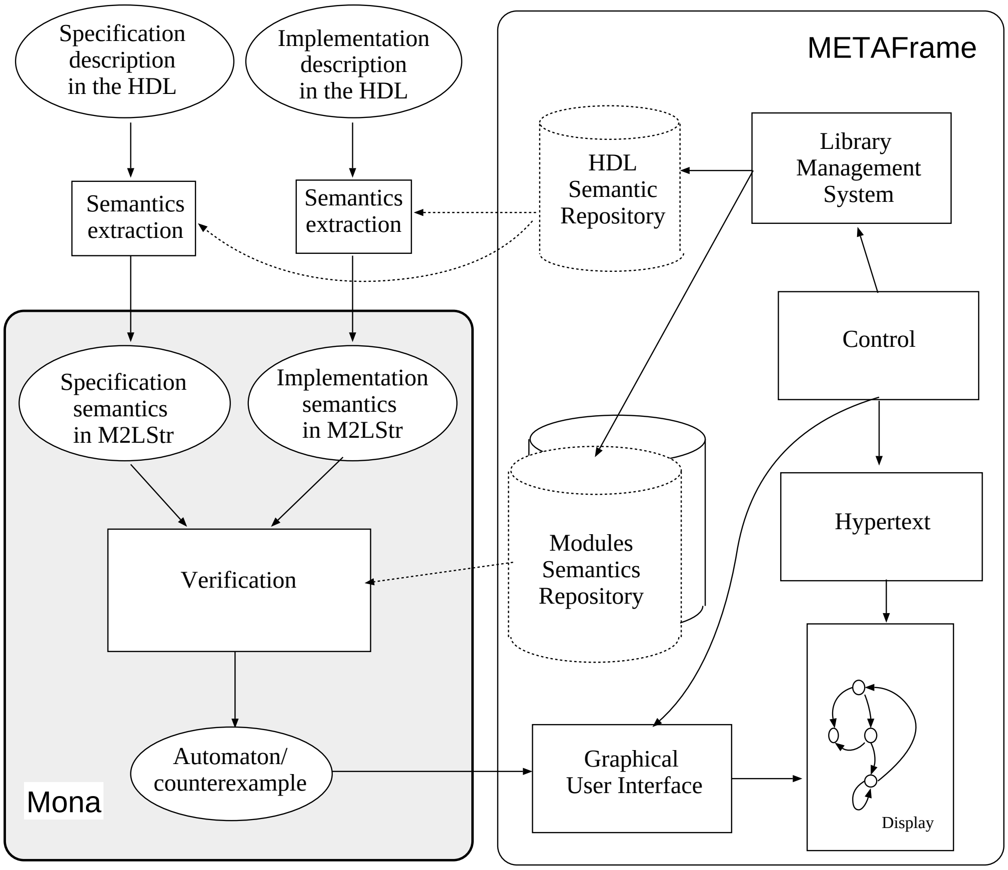


Figure 3: The verification scenario

```
All t: (t in Qn <=> t notin Q)))
```

giving rise to the following alternative definition of the D flip-flop:

```
dff(D,Q,Qn) = ((All t: t<$ => (t +o 1 in Q <=> t in D)) &
               Qn = compl(Q) &
               0 notin Q & 0 in Qn)
```

where we have used the equivalent formulation on second order variables. The equivalence of these formulations is easily proved automatically with Mona.

5 The Verification Environment

The architecture of our verification setup (illustrated in Figure 3) is based on the cooperation between Mona [HJJK95] and the MetaFrame environment [SMCB96, StMC95].

As circuit descriptions given in a Hardware Description Language can be first translated in the target logic, along the lines introduced in [Evek90, Marg91, Marg93][4], proofs can

[4]In these approaches the semantics of Register-Transfer and gate level descriptions was expressed in terms of first-order logic formulas.

be carried out fully automatically at the *logic* level by means of Mona. The result, in form of an automaton or of a minimal counterexample, can be visualized as a graph within the MetaFrame environment.

Mona is a verification tool for a second order monadic logic implemented in ML. Predicates are defined as logic formulas and automatically transformed into minimal automata. BDD techniques are used in order to store the automata's transitions. The computed automata are persistent objects within a user session, that is they are computed only once and stored in a library with their BDD structure. Later references to the predicate cause the precomputed automaton to be loaded and reused. Hence definitions constitute a (temporary) library of reusable components which remain available for later reuse within the same user session. The main advantage is the support of compositionality: since the automata constitute the semantics of the hardware objects, once the equivalence between two logic objects has been proved they become semantically indistinguishable. In particular, a specification can be then replaced by its implementation (compositional verification) and structural descriptions of a module can be replaced by their behavioral counterpart (mixed-mode verification).

The user may interact with both systems simultaneously as illustrated by Figure 4. Currently, MetaFrame (window (a)) provides the graphic facilities for the display of the results delivered by Mona (window (b)). Semantic models are automata, visualized as graphs: windows (c) and (d) show two of the automata generated by Mona for the case study of Sect. 6. MetaFrame also allows the user to investigate properties of the graphs by means of hypertext inspectors for nodes and edges: in addition to their label, nodes have in fact attributes like *start*, *accepting*, *non-accepting*, which are listed as shown in window (d) for the start node of automaton (c), and in window (f) for an edge of graph (e). Some of the properties shown in the inspector windows are also indicated by colouring of the nodes/edges in the graph.

6 Case Study: the Example of [RhSo93] Revisited

This section illustrates on an example the use of M2L(Str) as a powerful, elegant, and concise description language, which allows an easy description of VLSI devices with regular structure in a very compact form. Entire classes of systolic circuits are captured by a single parameterized description. The parameter can be interpreted as indicating the "*length*" of the device in terms of the number of cells. Specific instances can be modelled by simply specifying the actual value of the structural parameter.

6.1 The Original Statement of the Problem

The example chosen by Somenzi and Rho intends to investigate a case where two iterative systems are equivalent while their basic cells are not. They say:

> *The most interesting case is dealt with when two iterative systems with two non-equivalent basic cells become equivalent due to don't care sequences*[5].

[5]Don't care sequences form the language of the sequences that are never applied to a cell.

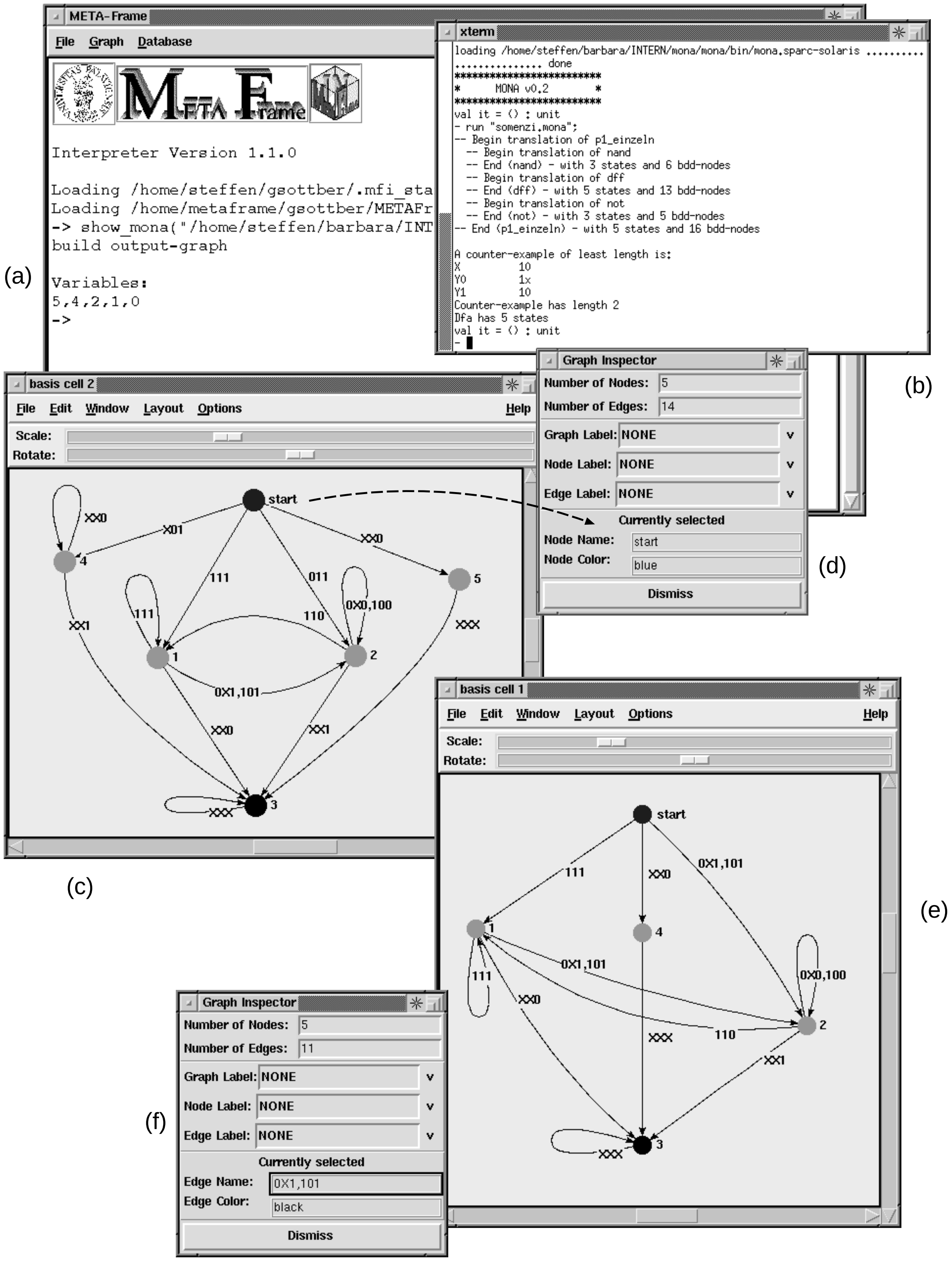


Figure 4: Snapshot of a user session

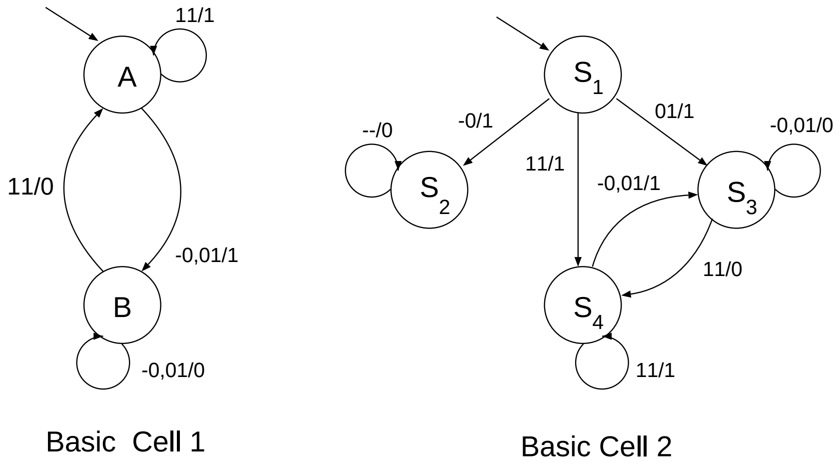


Figure 5: Automata of the Two Different Basic Cells

Fig. 5 shows two simple FSMs that are not equivalent. Nevertheless, we want to verify if two iterative systems with those FSMs as basic cells are equivalent. Note that this is not a case of different encodings of the same machine, since we start with two minimal deterministic FSMs and they have different numbers of states. Fig. 6 shows the gate-level implementation of the basic cell reported in [RhSo93]. Here the `X` input is considered to be local, and `Y0` a communication input. There are no local outputs, and a single communication output `Y1`. For the verification of the sequential iterative systems generated by these basic cells, the boundary condition of the first cell is set to `0`.

Accordingly, the following two verification problems will be addressed:

- **Problem 1:** *Verify that the sequential circuits implementing the basic cells are not equivalent.*

- **Problem 2:** *Verify that the iterative systems generated by the basic cells and under the given boundary condition are equivalent.*

The rest of this section is organized as follows: after a brief introduction of the gate-level library of predicates used for modelling the circuits (Sect. 6.2), for each of the above defined verification problems we will first present our modelling (Sect. 6.3 and 6.6) and then discuss how we handle the corresponding proofs (Sect. 6.4 and 6.7). We also show how to deal with partial verifications (Sect. 6.5) and with the verification of abstract properties (Sect. 6.8). Finally, we show in Sect. 6.9 an example of library-based verification.

6.2 The Gate-level Library

The following relations model in our logic the behaviour of the elementary gates.

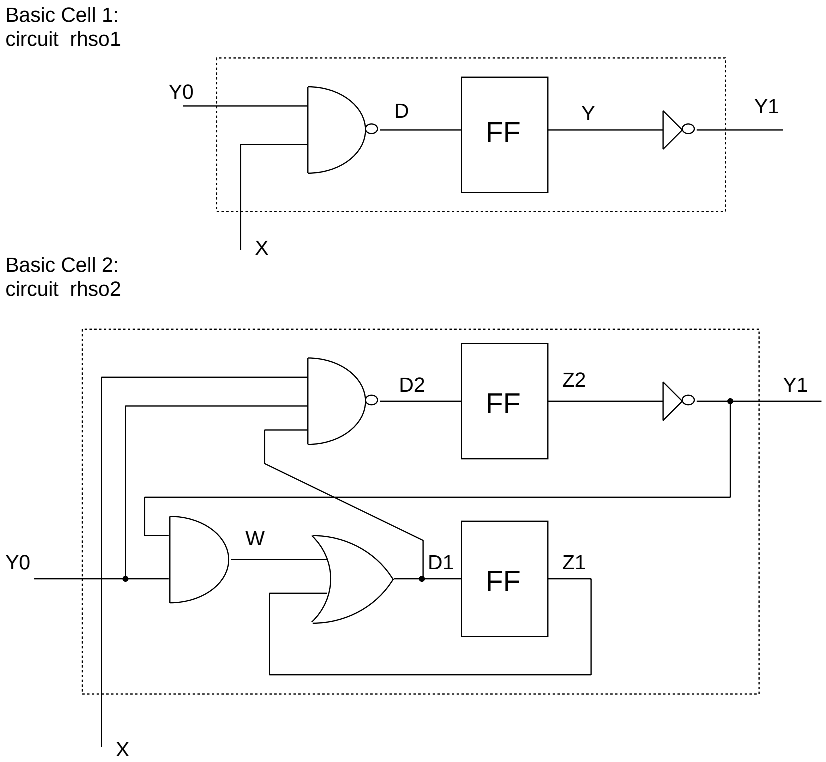


Figure 6: Original Implementation of the Basic Cells: Circuits `rhso1` and `rhso2`

Combinational circuits:

```
# circut constructors as relations
  not(@a,@b) = (~(@a) <=> @b);
  and(@a,@b,@c) = ((@a & @b) <=> @c);
  or(@a,@b,@c) = ((@a | @b) <=> @c);
  or3(@a,@b,@c,@d) = ((@a | @b | @c) <=> @d);
  xor(@a,@b,@c) = (((~@a & @b) | (@a & ~@b)) <=> @c);
  nand(@a,@b,@c) = ((~(@a & @b)) <=> @c);
  nand3(@a,@b,@c,@d) = ((~(@a & @b & @c)) <=> @d);
```

Note that we use relations instead of functions. This means that we are not primarily interested in modelling the signal flow in a circuit, distinguishing causes (variations on the inputs) and effects (the induced variations on the outputs), but rather *consistency conditions*. This is a more abstract view, which pays off when dealing with bidirectional signal flows (as also found in buses or at the transistor level in the pass transistors). This way it is immediately possible to model e.g. bilateral communication in systolic systems with no additional cost.

Sequential circuits:

They require the use of state-holding devices, the simplest of which is a simplification of the D flip-flop discussed in the previous section. Here the flip-flop is merely a delay:

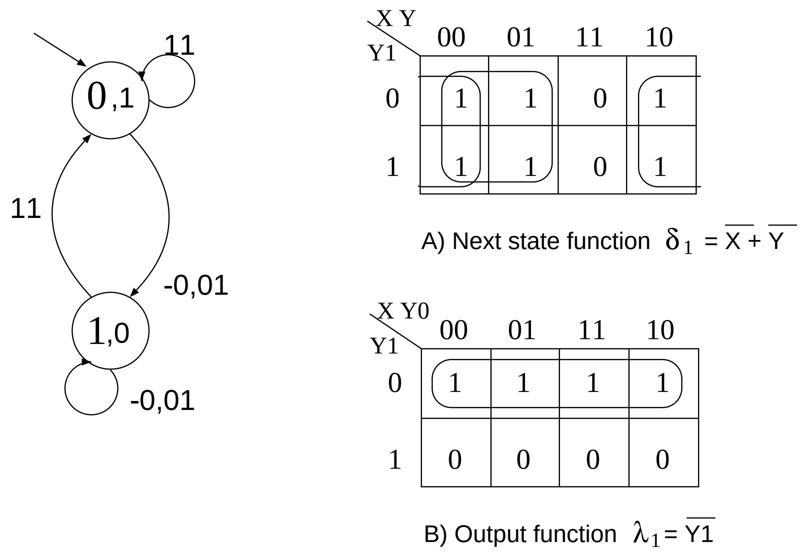


Figure 7: Basic Cell 1: Encoded State Diagram and Karnaugh's Maps

```
# D-type flip-flop
   dff(D,Q) = (All t: (t < $) => (t +o 1 in Q <=> t in D) &
                                   (0 notin Q));
```

6.3 Problem 1: The Basic Cells

Re-engineering the Original Implementation

Some reverse- and re-engineering is needed in order to discover the state assignments used by the authors in [RhSo93] for their implementation of the shown automata. It turns out that the following state encodings have been used:

State	Y
A	0
B	1

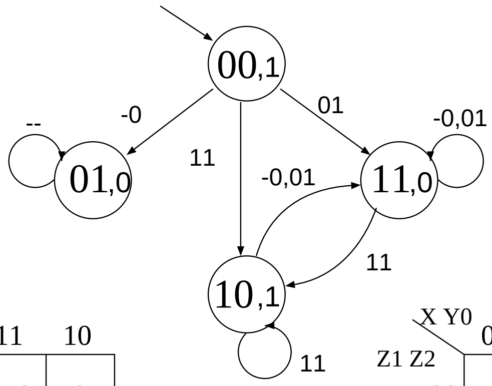
Encoding of Basic Cell 1

State	Z_1 Z_2
S_1	00
S_2	01
S_3	11
S_4	10

Encoding of Basic Cell 2

and that the circuits are indeed Moore machines, i.e., the output values are more appropriately written on the states. This gives rise to the encoded state diagrams and Karnaugh maps of Fig. 7 and 8, respectively. Here, transitions are labelled only with inputs, the outputs are written inside the states.

Re-synthesizing the basic cells by means of two-level techniques, we reach the following boolean expressions for the state transition and output functions:



X Y0 / Z1 Z2	00	01	11	10
00	01	11	10	01
01	01	01	01	01
11	11	11	10	11
10	11	11	10	11

A) Next state function δ_2

X Y0 / Z1 Z2	00	01	11	10
00	1	1	1	1
01	0	0	0	0
11	0	0	0	0
10	1	1	1	1

B) Output function λ_2

Figure 8: Basic Cell 2: Encoded State Diagram and Karnaugh's Maps

Basic Cell 1:

$\delta_1 = \overline{X} + \overline{Y_1}$

$\lambda_1 = \overline{S}$

Basic Cell 2:

$\delta_2 = (Z_1 + Y_0\overline{Z_2}, \quad \overline{X} + \overline{Y_0} + \overline{Z_1}Z_2)$

$\lambda_2 = \overline{Z_2}$

The corresponding circuits (which we call `our-cell1` and `our-cell2`), shown in Fig. 9, are different from the original ones. This opens an immediate additional verification problem: the pairwise equivalence of the implementations of the basic cells. This is formulated as:

- **Problem 1A:** *Verify that each pair of sequential circuits implementing the same basic cell is equivalent, i.e.,*

  `rhso1` $\Longleftrightarrow$ `our-cell1` *and* `rhso2` $\Longleftrightarrow$ `our-cell2`.

The Gate-Level Descriptions

The gate-level implementation of the original basic cells are given in the usual structural fashion, as netlists of gate-level components. Two new predicates `rhso1` and `rhso2` describe the basic cells as a collection of single gates with the appropriate connections. The body of each predicate is a conjunction of calls to predicates of the gate-level library, where the parameter-passing mechanism is used to establish the desired wiring. Internal connections (called nets) are hidden by means of existential quantification.

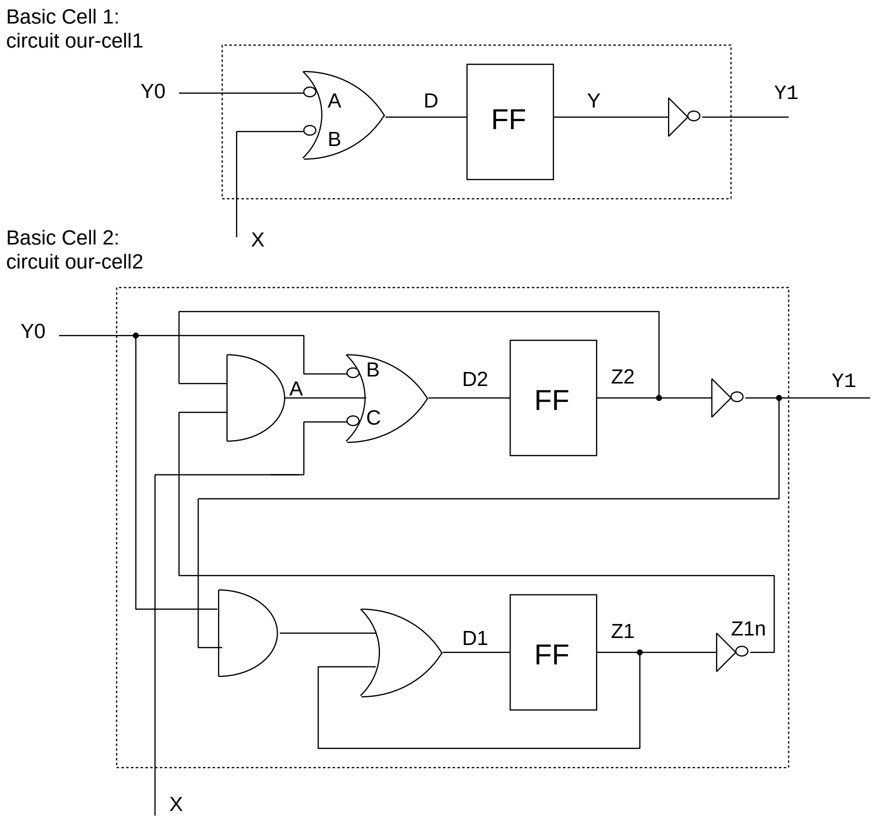


Figure 9: Re-implemented Basic Cells: Circuits `our-cell1` and `our-cell2`

```
# Basic Cell 1
  rhso1(X,Y0,Y1) = (Ex D: Ex Y: All t:
                    nand(t in X,t in Y0,t in D) &
                    dff(D,Y) &
                    not(t in Y,t in Y1));
# Basic Cell 2
  rhso2(X,Y0,Y1) = (Ex D1: Ex D2: Ex W: Ex Z1: Ex Z2: All t:
                    nand3(t in X,t in Y0,t in D1,t in D2) &
                    dff(D2,Z2) &
                    not(t in Z2,t in Y1) &
                    and(t in Y0,t in Y1,t in W) &
                    or(t in W,t in Z1,t in D1) &
                    dff(D1,Z1));
```

Problem 1A: Describing the New Implementations

The circuits of Figure 9 are easily described in our logic too. We give here the description of the first basic cell (predicate `our-cell1`) as sequential circuit

```
our-cell1(X,Y0,Y1) = (Ex A: Ex B: Ex D: Ex Y: All t:
                       not(t in Y0,t in A) & not(t in X,t in B) &
                       or(t in A,t in B,t in D) &
                       dff(D,Y) &
                       not(t in Y,t in Y1));
```

and of the combinational part of both versions of the second basic cell (predicates `rhso2comb`, corresponding to the original circuit, and `our-cell2comb`, corresponding to the new implementation)

```
rhso2comb(X,Y0,Y1,Z1,Z2,D1,D2) =
         (Ex W: All t:
          nand3(t in X,t in Y0,t in D1,t in D2) &
          and(t in Y1,t in Y0,t in W) &
          or(t in W,t in Z1,t in D1) &
          not(t in Z2,t in Y1));

our-cell2comb(X,Y0,Y1,Z1,Z2,D1,D2) =
             (Ex A: Ex Z1n: Ex B: Ex C: Ex W: All t:
              not(t in Z1, tin Z1n) &
              and(t in Z2,t in Z1n,t in A) &
              not(t in Y0, t in B) &
              not(t in X, t in C) &
              or3(t in B,t in A,t in C,t in D2) &
              and(t in Y1,t in Y0,t in W) &
              or(t in W,t in Z1,t in D1) &
              not(t in Z2,t in Y1));
```

which will be used to show different forms of equivalence.

6.4 Verification 1: Behaviour of the Single Cells

The Automata of the Original Cells

It is possible to generate automatically in Mona the automaton corresponding to each predicate. Considering the original cells, the following commands

```
rhso1(X,Y0,Y1)
rhso2(X,Y0,Y1)
```

cause Mona to compute the minimal automata corresponding to the predicates describing the original implementations of the basic cells. As usual, free variables are implicitely universally quantified.

The output offered by Mona consists of a listing of transitions presented in textual form, as shown in Fig. 10.

```
Automaton of Cell 1        Automaton of Cell 2

   (X,Y0,Y1)                  (X,Y0,Y1)   # Variables
   (4,2,1,0)                  (5,4,2,1,0) # Accepting states

   (0,4,0X0)                  (0,5,000)
   (0,2,0X1)                  (0,4,001)
   (0,4,100)                  (0,5,010)
   (0,2,101)                  (0,2,011)
   (0,4,110)                  (0,5,100)
   (0,1,111)                  (0,4,101)
   (1,3,0X0)                  (0,5,110)
   (1,2,0X1)                  (0,1,111)
   (1,3,100)                  (1,3,0X0)
   (1,2,101)                  (1,2,0X1)
   (1,3,110)                  (1,3,100)
   (1,1,111)                  (1,2,101)
   (2,2,0X0)                  (1,3,110)
   (2,3,0X1)                  (1,1,111)
   (2,2,100)                  (2,2,0X0)
   (2,3,101)                  (2,3,0X1)
   (2,1,110)                  (2,2,100)
   (2,3,111)                  (2,3,101)
   (3,3,XXX)                  (2,1,110)
   (4,3,XXX)                  (2,3,111)
                              (3,3,XXX)
   Dfa has 5 states           (4,4,XX0)
                              (4,3,XX1)
                              (5,3,XXX)

                              Dfa has 6 states
```

Figure 10: Mona's textual otput

A visualization in MetaFrame of the same automata is shown in Fig. 4 (e) and (c), respectively, after minimization of the number of transitions[6]. The visualized automata are indeed equivalent to the original ones, since the failure states (states 3 and 5, respectively) can be eliminated: they are in fact unreachable, being prohibited by the initialization of all the flip flops to 0. Fig. 11 shows the reachable portions of the automata after further minimization. They coincide with the automata of Fig. 5, our behavioural specification.

[6]As an example, the transitions (1,3,0x0), (1,3,100), and (1,3,110) of Basic Cell 1 are equivalent to the single transition (1,3, XX0), which is then displayed in Figure 4 (left).

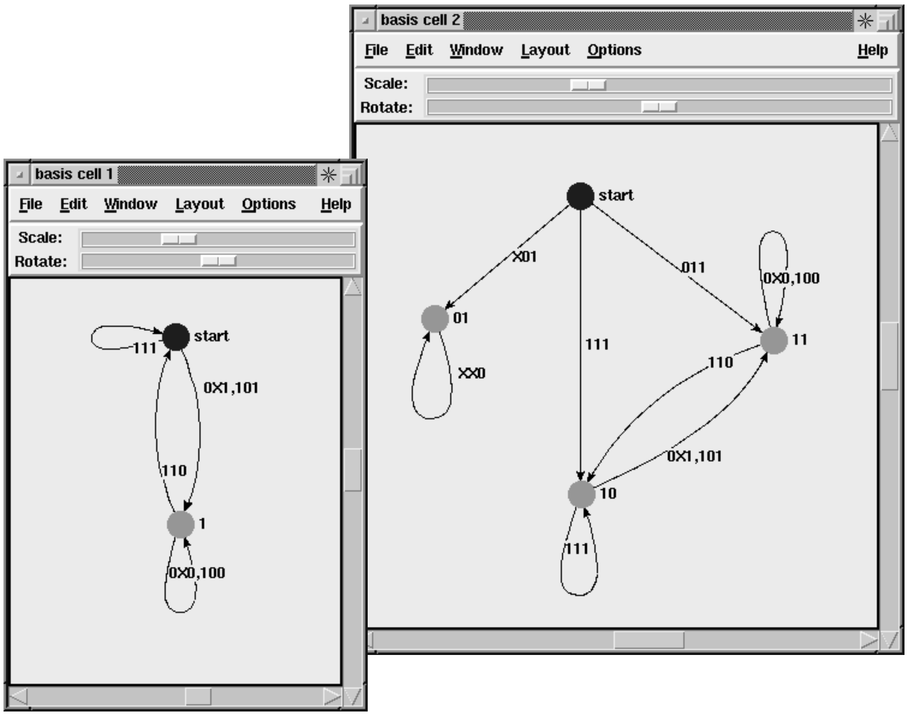


Figure 11: METAFrame's minimized automata

Equivalence of the two Original Basic Cells

The following formula states that the two original implementations of the basic cells, `rhso1` and `rhso2`, show the same behaviour over time, independently of the assignment to the free variables. Note that the same input `X` is fed to both circuits and that we expect to observe the same output sequences `Y0, Y1` at each pair of corresponding outputs.

```
rhso1(X,Y0,Y1) <=> rhso2(X,Y0,Y1)
```

This statement is false as expected, since the corresponding minimal deterministic automata (i.e. the two automata of Figures 5, or equivalently the two of Figure 11) are different.

6.5 Verification 1A: The Re-implemented Basic Cells

Considering now the alternative implementations of the basic cells at hand, i.e. the circuit pairs (`rhso1, our-cell1`) and (`rhso2, our-cell2`), we want to prove that each pair of circuits implementing the same basic cell is equivalent. To this aim, several approaches may be possible.

Complete Verification.
The straightforward approach to the problem would be proving the equivalence of each pair of circuits by solving a complete problem of sequential circuits verification, along

the lines of the previous section. In the case of Basic Cell 1, this leads to the following solution.

Basic Cell 1:
The equivalence of the two sequential circuits implementing Basic Cell 1, i.e.

```
rhso1(X,Y0,Y1) <=> our-cell1(X,Y0,Y1)
```

is proved to be a theorem by Mona in less than 1s CPU time. This result was expected, since both implementations have been derived from the same behavioural specification.

Partial Verification.
We may however use any additional information at hand in order to reduce the verification problem to its essential portion. To this aim, it is essential to know how to relate some aspects of both implementations. The following observations lead to increasing simplifications in the size of the verification problem:

1. From the Karnaugh maps we see that the output functions of each pair are identical. Moreover, the output portion of each circuit pair is indeed topologcally identical. The equivalence of each pair of circuits can thus be reduced to a proof of equivalence of the *state transition* portions as *sequential circuits*: if the circuits are in equivalent states, we already know that they have equivalent outputs.

2. We additionally know that the state encodings used for each pair of circuits are identical, therefore the verification of the *combinational* circuit determining the next state suffices.

3. Strictly seen, with a topological analysis it would be possible to limit the verification to those *portions* of the combinational circuits which differ in the two implementations. For Basic Cell 2, this would mean checking only `D2`, since the circuit delivering `D1` is exactly the same in both implementations. Since Basic Cell 1 has only one flip-flop, this third criterium does not apply.

Not having a topological analyzer for Mona, for Basic Cell 2 we resort to the verification of the full combinational portion, thus making use of observations (1) and (2). This leads to the following solution:

Basic Cell 2:
The equivalence of the combinational portions of the two implementations of Basic Cell 2 can be stated as

```
p2comb(X,Y1,Y2,Z1,Z2,D1,D2) <=> p2comb_imp(X,Y1,Y2,Z1,Z2,D1,D2)
```

Its proof is obtained automatically with Mona in less than 1s CPU.

6.6 Problem 2: The Iterative Linear Arrays

The implementation of the arrays generated by the described pairs of basic cells is obtained by appropriately connecting neighbouring instances of the basic cells along the communication channels. We intend to characterize the behaviour of the array as experienced by a signal entering the leftmost cell at time 0, and following its propagation along the array. To this aim, we define a second order variable UY, which describes what happens at each observation point between two adjacent cells: for both arrays, the Y1 output of the first cell is fed into the Y0 input of the adjacent one. The boundary condition for the first cell is enforced by saying that the initial value of the communication input Y0 is low (position 0 is not contained in the UY set). This connection scheme leads to the following descriptions:

```
# Systolic Array 1
  array1(X,Y) = (Ex UY: (0 notin UY) &
                        All t: ((t > 0) => (t in UY <=> t in Y)) &
                rhso1(X,UY,Y));

# Systolic Array 2
  array2(X,Y) = (Ex UY: (0 notin UY) &
                        All t: ((t > 0) => (t in UY <=> t in Y)) &
                rhso2(X,UY,Y));
```

The original statement of the verification problem in [RhSo93] was ambiguous wrt. the definition of the boundary condition, which concerns input Y0 of the first cell of the array: it says in fact that the boundary condition is 0, but it does not specify *when*. There are at least two legitimate interpretations:

- the Y0 signal of the first cell is *constantly* low (i.e. in our modelling language, UY = empty), or
- the Y0 signal of the first cell is low *only at initialization time*, i.e., in the first clock cycle, and then it is free to fluctuate. This is modelled in M2L(Str) as 0 notin UY.

We chose to follow the second interpretation, i.e. to apply the boundary condition only at initialization time, and then interprete the description as *following one family of computations* of the array. This means, for the initial computation we are "riding through the array" at the speed of the signals, entering the first cell at the initial time, and moving at each clock cycle to the right neighbour. The subsequent computations follow the same pattern, but the Y0 input of the first cell is no longer 0. Leaving the local inputs as free variables, we are able to capture all possible executions of the array (consistent with the initialization of the storage elements and with the initial boundary condition), modelled essentially in a *pipeline* fashion.

6.7 Verification 2: Behavioural Equivalence of the Arrays

The equivalence theorem is easily stated as the following M2L(Str) formula:

```
array1(X,Y) <=> array2(X,Y)
```

Mona constructs on-the-fly the minimized automaton corresponding to each predicate encountered during the expansion of the formula, as shown in the following screendump:

```
- run "somenzi.mona";
-- Begin translation of array1
  -- Begin translation of rhso1
    -- Begin translation of nand
    -- End (nand) - with 3 states and 6 bdd-nodes
    -- Begin translation of dff
    -- End (dff) - with 5 states and 13 bdd-nodes
    -- Begin translation of not
    -- End (not) - with 3 states and 5 bdd-nodes
  -- End (rhso1) - with 5 states and 16 bdd-nodes
-- End (array1) - with 4 states and 5 bdd-nodes
-- Begin translation of array2
  -- Begin translation of rhso2
    -- Begin translation of nand3
    -- End (nand3) - with 3 states and 7 bdd-nodes
    -- Load dff with 5 states and 13 bdd-nodes
    -- Load not with 3 states and 5 bdd-nodes
    -- Begin translation of and
    -- End (and) - with 3 states and 6 bdd-nodes
    -- Begin translation of or
    -- End (or) - with 3 states and 6 bdd-nodes
    -- Load dff with 5 states and 13 bdd-nodes
  -- End (rhso2) - with 6 states and 20 bdd-nodes
-- End (array2) - with 4 states and 5 bdd-nodes

Formula with free variables X Y is a tautology
Dfa has 1 states

 (X,Y)
 (0)
 (0,0,XX)

The total time elapsed: 0:00:02
val it = () : unit
```

The result, obtained in less than 2s CPU time, is the trivial automaton reported in Fig. 12 (left). It shows that the formula is indeed a tautology, since for any input the computation stays in the only state, which is accepting.

6.8 Verification of Abstract Properties

In addition to the verification of alternative implementations or of a specification/ implementation relation between two circuit descriptions, we can formulate and automatically verify relevant *abstract properties* of the systems under consideration. This feature is particularly interesting when handling complex parametric systems, like the iterative sequential systems we are studying.

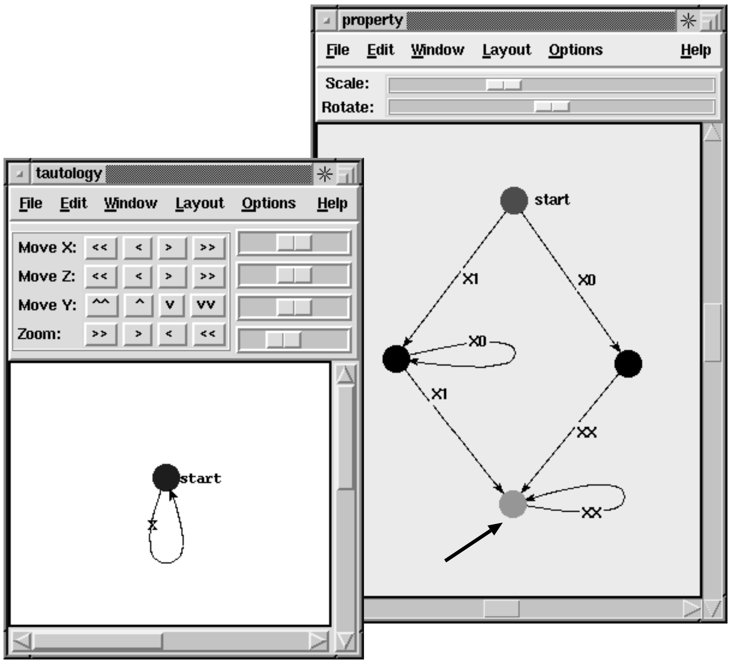


Figure 12: Verification results for the two arrays

A relevant property of iterative systolic arrays is e.g. the satisfiability of the formulas expressing their behaviour. The property is easily formulated as follows:

```
array1(X,Y) => false
array2(X,Y) => false
```

Here we prove satisfiability of a formula by computing a counterexample for its negation, a technique largely used in theorem proving and resolution-based provers.

The result is obtained automatically by Mona in less than 1s CPU. Neither formula is true, since both lead to the minimal automaton of Fig. 12 (right): the only accepting state, marked by the arrow in the picture (which would be red on the screen), is only reachable through non-accepting states. This proves that the behaviour of the arrays is indeed satisfiable.

6.9 Library-based Verification

Currently, the internal representation of already processed M2L(Str) predicates is retained by Mona for the duration of a tool session. This means that issuing the same command – `run "somenzi.mona";` – a second time will not cause the recomputation of all predicates from scratch. Instead, those predicates which have not been modified since the last issue of the same command are still available in a (temporary) library of minimized automata in BDD format, and are just loaded. In our example, a second request of verification causes the following reaction:

```
- run "somenzi.mona";
-- Load array1 with 4 states and 5 bdd-nodes
-- Load array2 with 4 states and 5 bdd-nodes

Formula with free variables X Y is a tautology
Dfa has 1 states

The total time elapsed: 0:00:00
```

No modifications of the predicates have occurred, therefore the two outermost predicates are still valid in the already computed form, and need just be re-loaded for the evaluation of the theorem. This feature is of vital importance for the verification of larger circuits, as demonstrated in [Marg96] for a hierarchical parametric family of synchronous counters. The basic drawback of the current implementation of this feature of Mona is that the lifespan of the computed automata is limited to a single user session, as discussed in the next section.

7 Lessons Learned

The outcome of this case study has brought clarity on several issues, some of which are quite unexpected.

7.1 The Language

The chosen formalism, although quite unfamiliar at first, has proved to be fairly intuitive to use even for newcomers with little experience in hardware design and in logics. However, a possibility of input directly from a textual description given in some Hardware Description Language or from a circuit drawing would be a helpful improvement.

7.2 Modelling Hardware in this Logic

The expressive power of M2L(Str) captures only one-dimensional structures (linearly or circularly arranged). This is due to the interpretation of the logic over strings, which implies that the parameterization allowed to express generalized behaviours is limited to the generic "length" of strings. Since strings may be taken to assume different meanings (here and in [MaMe96] sampled waveforms, in [Marg96] the bitwidth of a datapath), a degree of freedom is left to the designer.

The bad news are that one in general needs to talk over behaviours of classes of circuits over time, which calls for genericity along *both* time and structural dimensions. It is possible to some extent to 'cut' along one of the two axes: here we have renounced to model the whole array at all times, and have chosen to model one (however generic) computation. In this case this choice still suffices to capture the whole behaviour, but this is not true in general.

An interpretation of basically the same logic over trees is still decidable, and we know of two implementations of it [7], but the tools are still rather prototypical, and basically lack documentation.

7.3 Evaluation of the Tool

Mona in its current version is still a research prototype, which needs a lot of getting used to. Its big merit lies definitely in its existence. This suffices for the purpose of demonstrating on several interesting case studies, spanning diverse application fields, that practical examples are indeed in general much better behaved than the staggering theoretical worst case complexity. However, the initial enthusiasm cools quickly down as soon as one starts a serious hands-on experience. Its most evident drawbacks are:

- Unclear *semantics* of the logic, which has never been formally defined, and unclear correspondence between the published and the implemented versions of the logic. Problems are basically connected with the length of strings, and we have documented difficulties with all the related operators. In fact, not all equivalent formulations of predicates and theorems lead to the expected results. Careful placement of parenthesis and extra assumptions are sometimes necessary in order to avoid problems with empty strings[8]. The confidence in the results cannot therefore be blind.

- Rigid *user interface* of the tool, which is in pure textual form. Mona accepts only M2L(Str) formulas and delivers automata and counterexamples only as list of transitions, as shown in Section 6. Since the tool is basically a translator into finite automata, it is immediate to wish a the possibility of direct input of an automaton (e.g. in order to use the tool as a model checker, which checks whether a formula holds for a given automaton), and some form of visualization of the generated automata Not even the same format for automata descriptions delivered as output can be read and used again by the tool!

- Shallow *integrability* of the tool in larger environments. The initial enthusiasm about the embedding of Mona into MetaFrame was soon frustrated by the rigid interface of the first. The embedding had in fact to be limited to a shallow form of one-directional cooperation. Since Mona has to run inside the ML interpreter, it was not possible to launch it from the MetaFrame environment, with the consequence that we could not use Mona as planned, i.e., as an external decision procedure callable at need with a simple command.
  The powerful graph manipulation and the wealth of available analysis and verification tools of MetaFrame was of no use too, given Mona's incapability of loading anything else than M2L(Str) formulas. Conversely, displaying and automatically layouting Mona graphs was a matter of a day in MetaFrame. Those graphs can in fact be edited and reused by the other tools of the MetaFrame environment.

[7]A Master Thesis in Kiel, in W. Thomas' group and the Mona++ package from Aarhus, by the same authors of the tool we used.

[8]The logic is defined only for non-empty strings [HJJK95].

- Very limited *reuse* of work. We have shown that the repository of automata created by Mona has the lifespan of a single session. Leaving the tool means losing the library, which is an unexpected waste of computation time for a tool implementing a non-elementary decision procedure. Many application domains are solidly library-based; e.g., the complexity of hardware design lives out of a multi-layered hierarchy of circuit libraries, which are defined for different abstraction levels (e.g., mask, transistor, gate, register-transfer). They contain circuits which are "certified" correct only once and then reused innumerable times. It is inconceivable to have to recompute their semantics every time the system is started! Not having access to the source files of Mona we had no possibility of eliminating this serious drawback.

7.4 Understanding of this Case Study

The modelling of the example in the logic has also brought us more clarity about what really happens in the considered example.

Rho and Somenzi say that the two iterative systems are equivalent while their basic cells are not, and spot (correctly) the reason in the language of the sequences that are *never applied* to the system.

What they in fact mean is that the *context* in which the arrays are put, defined (i) through the given boundary condition to the array and (ii) through the initialization conditions of the sequential circuits implementing the basic cells, forbids behaviours which may distinguish the two systems.

According to our formulation of the verification problems (Sect. 6.1), the equivalence of the basic cells and the equivalence of the iterative systems are problems of two different natures, whose difference is much clearer in our M2L(Str) modelling than in the BDD-based formulation proposed in their work.

In our model, the verification of the basic cell concerns a verification problem of a *control automaton*, which is a typical sequential problem. In this case, both the external and the communication channels are free inputs and outputs, and are allowed to float freely. We observe only the behaviour over time, comparing it from an initialization state in which the memory elements are reset to 0.

The verification of the array is concerned with a very different problem, which interests the connections along the iterative structure. The problem is much more similar to a *data-path* verification problem, where one observes the propagation of signals along a structure. Here only the cell's external ports may float, the communication channels are now the propagation structure and we observe a behaviour along those channels, comparing it starting with an initialization state determined by the boundary condition (initially 0 on the left end of the array).

This is a major additional restriction wrt. to the context considered for the verification of the single cells, since it eliminates half of the possible behaviours from our observation. Indeed it is possible to say that the *now* eliminated behaviours *don't care*, and that they form the language of the never applied sequences.

The modelling drawbacks presented by Rho and Somenzi's method are due to the use of the same automata-based modelling for both classes of problems. The need of reasoning directly on the automata manipulations tends to obscure the phenomena and the nature of complex problems. Their approach was in fact originally designed as a computationally efficient means for handling single sequential circuits. It can accommodate the treatment of some classes of iterative parametric combinations of sequential circuits, but at the price of a very technical modelling. In this respect some abstraction, like the one forced by using our intermediate logic, is often an advantage.

8 Conclusions

This paper presented the use of M2L(Str) in the description of a particular class of VLSI devices, namely iterative linear systolic arrays. The formalism is modular and hierarchical. Recursivity over time is implied in the description of sequential behaviour through difference equations. Parametrization also eases the description of regular structures. Descriptions are verification-oriented, although they are here presented in a self-standing form.

The implementation in the MetaFrame environment of a new decision procedure for essentially the same logic is about to start.

Acknowledgements

The author is indebted to Claudia Gsottberger and to Michal Mendler for many fruitful discussions and for their valuable help in the realization of this case study.

References

[Alle94] M. Allemand: "*Formal verification of characteristic properties*: Proc. TPCD'94 (Theorem Provers in Circuit Design - Theory, Practice, and Experience), Bad Herrenhalb (D), Sept.'94, LNCS N. 901, pp. 292-297.

[AnCD91] C. Angelo, L. Claesen, H. De Man: "*A Methodology for Proving Correctness of parameterized hardware Modules in HOL*," Proc. CHDL'91, Marseille (F), April 1991, IFIP Transactions, North-Holland, pp.63-82.

[BaKl95] D. Basin, N. Klarlund: "*Hardware verification using monadic second-order logic*," Proc. CAV '95, Liège (B), July 1995, LNCS N. 939, Springer Verlag, pp. 31-41.

[BaDV89] D. Basin, P. DelVecchio: "*Verification of combinational logic in Nuprl,*" In "Hardware Specification, Verification and Synthesis: Mathematical Aspects", Ithaca, New York, 1989. Springer-Verlag.

[BrCG89] M. Browne, E. Clarke, O. Grumberg: "*Reasoning about networks with many identical finite state processes*," Information and Computation, 81(1), Apr. 1989, pp. 13-31.

[CaGM87] A. Camilleri, M. Gordon, T. Melham: "*Hardware verification using higher-order logic*," In D. Borrione (ed.),"From HDL Descriptions to Guaranteed Correct Circuit Designs", pp. 43–67. Elsevier Science Publishers, (North-Holland), 1987.

[CaMP89] P. Camurati, T. Margaria, P. Prinetto "*Systolic array description in F^2*," Microprocessing and Microprogramming, The Euromicro Journal, Vol. 27, n. 1-5, Sept. 1989, pp. 171-178.

[Chur63] A. Church: "*Logic, arithmetic and automata*," Proc. Int. Congr. Math., Almqvist and Wiksells, Uppsala 1963, pp. 23-35.

[Cons86] R. L. Constable et al.: "Implementing Mathematics with the Nuprl Proof Development System," Prentice-Hall, Englewood Cliffs, NJ, 1986.

[Evek90] H. Eveking: "*Axiomatizing Hardware Description Languages*," Int. Journal of VLSI Design, 2(3), pp. 263-280, 1990.

[Gord86] M. Gordon: "*Why higher-order logic is a good formalism for specifying and verifying hardware*," In G. J. Milne and P. A. Subrahmanyam, editors, "Formal Aspects of VLSI Design", North-Holland, 1986.

[GuFi93] A. Gupta, A. Fisher: "Parametric Circuit Representation Using Inductive Boolean Functions," Proc. CAV'93, Elounda (GR), June 1993, LNCS N. 697, pp.15-28.

[HaDa86] F.K. Hanna, N. Daeche: "*Specification and verification using higher-order logic: a case study*," In G.J. Milne, P.A. Subrahmanyam, eds., "Formal Aspects of VLSI Design", pp. 179–213. Elsevier, 1986.

[Hunt89] W. Hunt: "*Microprocessor design verification*," Journal of Automated Reasoning, 5(4):429–460, 1989.

[HJJK95] J. Henriksen, J. Jensen, M. Jørgensen N. Klarlund, R. Paige, T. Rauhe, A. Sandholm: "*Mona: Monadic second-order logic in practice*," Proc. TACAS'95, Aarhus (DK), May 1995, LNCS 1019, Springer Verlag, pp. 89-110.

[Koha70] Z. Kohavi: "Switching and finite automata theory", Computer Science Series, McGraw Hill, New York, NY (USA), 1970.

[KuMi89] R. Kurshan, K. McMillan: "*A structural induction theorem for processes*," Proc. 8th ACM PODC Sympos., Edmonton (CAN), Aug. 1989, pp. 239-247.

[KuLe79] H. T. Kung, C. E. Leiserson: "Systolic arrays (for VLSI)," Sparse Matrix Proceedings 1978, I. S. Duff and G. W. Stewart eds., Society for Industrial and Applied mathematics, 1979, pp. 256-282.

[Leis82] C. E. Leiserson: "Area efficient VLSI computation," Ph.D. Thesis, Carnegie-Mellon University, 1982.

[MaMe96] T. Margaria, M. Mendler: "Automatic Treatment of Sequential Circuits in Second-Order Monadic Logic", 4th GI/ITG/GME Worksh. on Methoden des Entwurfs und der Verifikation digitaler Systeme, Kreischa (D), March 1996, pp. 21-30, Shaker Verlag.

[Marg91] T. Margaria: "First-Order theories for the verification of complex FSMs," Aachener Informatik-Berichte Nr.91-30, RWTH-Aachen, Dec. 1991.

[Marg92] T. Margaria: "Efficient RT-Level Verification by Theorem Proving", IFIP World Congr.'92, Madrid (E), Sept. 1992, North-Holland pp. 696-702.

[Marg93] T. Margaria: "*Verifica formale della correttezza del progetto di sistemi digitali*," Dissertazione di Dottorato di Ricerca in Ingegneria Informatica e dei Sistemi (in Italian), Politecnico di Torino, Turin (I), Feb. 1993.

[Marg96] T. Margaria: "*Fully Automatic Verification and Error Detection for Parameterized Iterative Sequential Circuits*," Proc. TACAS'96, Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems, Passau, March 1996, LNCS N. 1055, Springer Verlag, pp. 258-277.

[MaSt93] T. Margaria, B. Steffen: "Distinguishing Formulas for Free", EDAC–EUROASIC'93: IEEE European Design Automation Conference, Paris (France), February 1993.

[Melh88] T. Melham: "*Using recursive types of reasoning about hardware in higher order logic*," In Int. Working Conf. on The Fusion of Hardware Design and Verification, pp. 26–49, July 1988.

[Pier89] L. Pierre: "The Formal Proof of the "Min-max" sequential benchmark described in CASCADE using the Boyer-Moore theorem prover," Proc. IMEC-IFIP Worksh. on Applied Formal Methods for Correct VLSI Design, Leuven (B), Nov. 1989, pp. 129-149.

[Pier94] L. Pierre: "*An Automatic Generalization Method for the Inductive Proof of Replicated and Parallel Architectures*," Proc. TPCD'94, Bad Herrenhalb (D), Sept.'94, LNCS N. 901, pp. 72-91.

[RhSo92] J.K Rho, F. Somenzi: "Inductive Verification for Iterative circuits", Proc. DAC'92, Anaheim (CA), June 1992, pp. 628-633.

[RhSo93] J.K Rho, F. Somenzi: "Automatic Generation of Network Invariants for the verification of Iterative sequential Systems", Proc. CAV'93, Elounda (GR), June 1993, LNCS N. 697, pp.123-137.

[SiGe87] A. Sistla, S. German: "*Reasoning with many processes*," Proc. LICS'97, Ithaca, NY, June 1987, pp. 138-152.

[SMCB96] B. Steffen, T. Margaria, A.Claßen, V.Braun: "*The* METAFrame*'95 Environment*" (Experience Report for the Industry Day), Proc. CAV'96, Int. Conf. on Computer-Aided Design, Juli-Aug. 1996, New Brunswick, NJ, USA, LNCS 1102, pp. 450-453, Springer Verlag.

[StMC95] B. Steffen, T. Margaria, A. Claßen. "The META-Frame: An Environment for Flexible Tool Management," Proc. TAPSOFT'95, Aarhus (Denmark), May 1995, LNCS N. 915, Springer Verlag.

[SuMe77] I. E. Sutherland, C. A. Mead: "Microelectronics and computer science," Scientific American, Vol. 237, N. 3, September 1977, pp.210-228.

[Thom90] W. Thomas: "*Automata on infinite objects*," In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, vol. B, pp. 133–191. MIT Press/Elsevier, 1990.

[Wolp86] P. Wolper: "*Expressing interesting properties of programs in propositional temporal logic*," Proc. POPL'86, St. Petersburg, Jan. 1986, pp. 184-192.

[WoLo89] P. Wolper, V. Lovinfosse: "*Verifying properties of large sets of processes with network invariants*," Proc. Automatic Verification Methods for Finite Systems, LNCS 407, Springer Verlag, 1989, pp. 68-80.