LIF

Laboratoire d'Informatique Fondamentale
de Marseille

Unité Mixte de Recherche 6166
CNRS - Université de Provence - Université de la Méditerranée

Certifying Circuits in Type Theory

Solange Coupet-Grimal, Line Jakubiec

Rapport/Report 03-2002

30 Mai 2002

Certifying Circuits in Type Theory

Solange Coupet-Grimal[1], Line Jakubiec[2]

Laboratoire d'Informatique Fondamentale

UMR 6166
CNRS - Université de Provence - Université de la Méditerranée

[1]CMI - Université de Provence, 39 rue Joliot-Curie,
F-13453 Marseille Cedex-13
[2]Faculté des Sciences de Luminy, 163 Av. de Luminy,
F-13288 Marseille Cedex-9

[1]Solange.Coupet@cmi.univ-mrs.fr, [2]Line.Jakubiec@lim.univ-mrs.fr

Abstract/Résumé

We investigate how to take advantage of the particular features of the Calculus of Inductive Constructions in the framework of hardware verification. First, we emphasize in a short case study the use of dependent types and of the constructive aspect of the logic for specifying and synthesizing combinatorial circuits. Then, co-inductive types are introduced to model the temporal aspects of sequential synchronous devices. Moore and Mealy automata are co-inductively axiomatized and are used to represent uniformly both the structures and the behaviors of the circuits. This leads to clear, general and elegant proof processes as it is illustrated on the example of a realistic circuit: the ATM Switch Fabric. All the proofs are carried out using Coq. *Keywords*: formal methods, hardware verification, type theory, dependent types, co-induction, extraction.

Nous étudions comment tirer parti au mieux, dans le domaine de la vérification de hardware, des particularités du Calcul des Constructions Inductives. Nous nous intéressons tout d'abord, dans une courte étude de cas, aux types dépendants et à l'aspect constructif de la logique sous-jacente pour spécifier et synthétiser des circuits combinatoires. Puis nous étendons notre étude aux circuits séquentiels synchrones en introduisant des types co-inductifs pour modéliser les aspects temporels. Nous donnons une axiomatisation co-inductive des automates de Moore et de Mealy que nous utilisons pour représenter uniformément les structures et les comportements. Il en résulte des processus de preuve clairs, généraux et élégants comme nous l'illustrons sur l'exemple d'un vrai circuit: l'ATM Switch Fabric. Toutes les preuves sont vérifiées à l'aide de l'assistant de preuve Coq. *Mots clés*: méthodes formelles, vérification de hardware, théorie des types, types dépendants, co-induction, extraction.

Relecteurs/Reviewers: Roberto Amadio, Sylvano Dal Zilio

1 Introduction

In recent years formal methods have been used increasingly in the verification of circuit designs and have appeared to be a good alternative to test and simulation techniques which present the drawback of being non exhaustive. Among them, model checking methods based on BDDs [Bry86] have the advantage to be fully automated. However they can lead to a combinatorial explosion of the state space and moreover they are not generic since the verification is for circuits of fixed size.

Formal specification combined with mechanical verification is a profitable approach for achieving the high levels of assurance required of safety-critical digital systems. In this paper we present a study for specifying and verifying circuits in Type Theory, and more precisely in the Calculus of (Co)-Inductive Constructions (CC). The motivation for choosing this logical framework is threefold. First, its great expressiveness makes it possible to give clear, accurate and generic specifications, to reason elegantly on them, and to obtain general and reusable results. Second, it rests on firm logical foundations. Third, it is implemented as a proof assistant, the Coq system [Tea01]. The latter, as it relies on a small kernel of rules, can be regarded as very reliable. Thus, we investigate thoroughly how to take advantage of the particular features of CC (dependent types, higher order logic, inductive and co-inductive types, extraction) in the framework of hardware verification. All the proofs are carried out using Coq.

The first part of this work deals with combinatorial circuits and their representation with dependent types. It is illustrated by a short case study based on the work done in [HDL90] and related to linear arithmetic architectures. It also presents a method for synthesizing this kind of devices by using the possibility to extract the informative content of λ-terms [CGJ96].

Then, this study is extended to sequential synchronous circuits. In addition to dependent types that we still use to give a precise description of their combinatorial parts, we introduce co-inductive types to take into account the temporal aspects of their specification.

Starting from a co-inductive representation of the history of the values carried by the wires, we model uniformly by means of Mealy and Moore automata both the structures and the behaviors. These two notions are thus considered as descriptions of the same entity, represented at different abstraction levels. Automata are axiomatized as fixpoints, representing non-ending processes that compute an infinite output sequence in response to an infinite input sequence. The set of automata is equipped with algebraic composition rules and with an equivalence relation which is proved to be a congruence for the composition rules. This makes a hierarchical approach possible since, in a modular device, a pre-proven sub-component can be replaced by its expected behavior. This axiomatization includes a co-inductive theorem about automata equivalence from which follow all the correctness proofs in the practical cases. This theorem captures once and for all the temporal aspects of the proofs, that are thus clearly separated from the combinatorial parts. This makes the proof process general, efficient and elegant.

We demonstrate the feasibility of this methodology on the example of a realistic non trivial circuit, namely the Fairisle ATM Switch Fabric. This circuit has been designed, built, and used at the University of Cambridge [LM91] [LM90] and has been widely used as a benchmark by the international hardware community. In spite of the complexity of the device, and due to the abstract description of automata, we obtain very compact representations (at most five states) with high level transition functions on rich data types [CJ99].

This paper is self-contained and organized as follows. Section 2 is a brief introduction to the Calculus of (Co)-Inductive Constructions. In section 3, we present on the example of a comparator, how to specify accurately linear arithmetic structures using dependent types and how to synthesize them using the Coq extraction mechanism. Section 4 is dedicated to axiomatizing automata. Then, in section 5, we present an application of our methodology to the ATM Switch Fabric. Finally, in section 6, we compare our study to other related work and we conclude in section 7.

2 A Brief Overview of the Calculus of Co-Inductive Constructions

The Calculus of Constructions (CoC) has been introduced by T.Coquand and G.Huet [CH85] and enriched with inductive types by C.Paulin-Mohring [PM96] and co-inductive types by E. Giménez [Gim96]. Simultaneously, successive versions of the system were implemented, resulting in the proof assistant Coq [Tea01].

The Calculus of Constructions

The Calculus of (Co-)Inductive Constructions (CC) is a higher order λ-calculus which provides in an uniform logical framework both a functional programming language for specifying and a higher order intuitionistic logic for reasoning about specifications, via the Curry-Howard isomorphism. The system relies on a dual interpretation of types, either as sets or as propositions. Thus a term `t`, inhabitant of a type `A`, can be both interpreted as an element of the set `A` and as a proof of the proposition `A`.

CC is obtained by enriching the simply typed λ-calculus in the following way. First, one allows types to depend on terms. For that, a sort `*` is introduced and `A:*` means "`A` is a type". New type constructors such as `P:A→*` can be interpreted both as predicates over the set `A` or as a family of sets, indexed by `A`. One can then construct types `(P a)` depending on terms `a` of type `A`. Under the assumptions `x:A, h:(P x)` one can derive $\vdash$ `λx:A.h : (∀x:A)(P x)`. The type `(∀x:A)(P x)` denotes either the product of a family of sets indexed by `A` or a universally quantified proposition.

Polymorphism is also introduced by abstracting terms with respect to types. One can then define the polymorphic identity:

`(λA:*)(λx:A)x:(∀A:*)A→A`

as well as the conjunction `A∧B` of two propositions `A` and `B` by the type:

`(∀C:*)(A→B→C)→C`.

Finally, abstracting types with respect to types is also possible. The conjunction ∧ for example, can be defined in the following way:

`∧ := (λA:*)(λB:*)(∀C:*)(A→B→C)→C`

Extraction

As the underlying logic is constructive, proofs are effective. This implies in particular that any proof of a statement of the form: `(∀x:A)(∃y:B)(P x y)` is a pair `(f:A→B, p)` where `f` is a program that, for all `x:A`, computes a witness `y=(f x)` and `p` is a proof of the proposition: `(∀x:A)(P x (f x))`. In this sense, such a term can be viewed as a certified program since `f` is accompanied with a "certificate" `p` ensuring that it meets some property. It is then interesting to be able to erase the logical part of the proof term, exactly as a compiler ignores comments, and, in such a way, to get the purely computational component `f` which has been proved to be correct. But this is meaningless in such a system as the one we have presented, since proofs and programs are undifferentiated. These considerations lead to splitting the sort `*` into two twin sorts called `Set` and `Prop`, enforcing the distinction between the two possible type semantics. This is how the system Coq can provide a mechanism which automatically extracts from a term the certified computational part.

Induction, Co-Induction

The calculus also makes it possible to define inductive and co-inductive types. Such a type is characterized by a finite set of typed constructors. Each constructor corresponds to an introduction rule of the underlying natural deduction system. As an illustration, let us review the various notions of *lists* that can be defined in the system.

Let `A` be a parameter of type `Set`, one can define the following types:

```
Inductive list:Set := nil:list | cons:A→list→list
Inductive d_list:nat->Set:=
     d_nil:(d_list 0) | d_cons:(n:nat)A→(d_list n)→(d_list (n+1))
CoInductive c_list : Set :=
     c_nil : c_list | c_cons : A→c_list→c_list
CoInductive Stream : Set := Cons : A→Stream→Stream
```

An induction principle is associated with each inductive type (and is automatically generated by the system Coq). It corresponds to an elimination rule for reasoning on the terms in the free algebra generated by the constructors. The

inductive type `list` represents the set of finite lists the elements of which are in `A`. If `n` is of type `nat`, `(d_list n)` is the dependent type of length-`n` lists. Total functions can be defined over inductive types by structural recursion.

In this study, we make heavy use of dependent lists, to give a precise encoding of circuit ports. The number of input and output wires is verified when type-checking the specifications. This prevents from tackling the proof process with erroneous or unreliable descriptions of the circuit.

In a dual way, the co-inductive type `c_list` corresponds to a greatest fixpoint, and denotes the set of all the finite and infinite lists of elements in `A`. The type `Stream` represents infinite sequences of such elements. The proofs of co-inductive statements are co-recursive terms. They are interpreted by fixpoints which represent non-ending processes that build infinite objects step by step. The co-recursive terms must meet a guard condition to be well-formed: a recursive call, within such a term, must occur just under a constructor.

Our methodology for verifying synchronous sequential circuits relies on encoding by streams the history of the values carried by the wires. In this way, a register, for example, is a function consing its initial value to its input stream. In the rest of this paper we shall write σ_0 for the head of a stream σ and σ' for its tail. Moreover, we shall use a predicate $\sim$ that represents the extensional equality over the streams and which is defined co-inductively by:

```
CoInductive ∼:  Stream→Stream→Prop :=
          eqS:(∀σ:Stream)(∀τ:Stream) σ0=τ0 → σ'∼τ'→ σ ∼ τ
```

Moreover, let us mention that these types depend on parameter `A`. For instance, the type of streams of elements of type `A` is in fact `(Stream A)`.

The following section handles combinatorial circuits and is a short case study that exemplifies the use of dependent lists and of the Coq extraction mechanism for specifying and synthesizing a certain class of arithmetic circuits. Then we present a more significant investigation, involving a co-inductive axiomatization of automata, a hierarchical methodology for verifying sequential synchronous circuits and the certification of a rather complex device.

3 Synthesizing Linear Architectures: Extraction and Dependent Types

This section illustrates on the case of linear arithmetic architectures how a certified circuit can be synthesized from its behavioral specification in type theory. It also exemplifies the use of sub-types and dependent types for accurate specifying.

Let us consider, for instance, the comparator depicted in fig.1. It is a hardware device that accepts two numerals and determines their relative magnitude. It is composed of identical cells interconnected by a carry wire accepting comparison

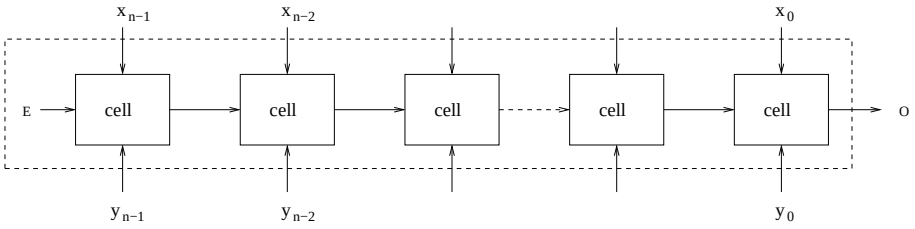

Figure 1: A Comparator

data in a 3-valued type. Each cell, from left to right, outputs a value that depends on the incoming carry and the result of the comparison of two digit inputs.

Our axiomatization makes heavy use of sub-types and dependent polymorphic lists that allow us to get high-level abstract specifications, more general than those in previous work on dependent types [HDL90]. This latter type is particularly suitable in the framework of hardware specification where linear structures are prevalent.

3.1 Axiomatizing numeration systems

The numeration system axiomatization is parameterized by its base which is represented as a term `BASE` of type

`BT:={b:nat|0<b}`.

`BT` stands for the type of strictly positive natural numbers. This type is defined inductively and the terms inhabiting it are pairs consisting of a natural number `b` and a proof that `b` is greater than 0. The natural injection from `BT` to the natural numbers is defined by associating with such a pair its first component. Let us call `base` the result of this injection on `BASE`.

The type of digits can then be defined similarly as the type of the natural numbers less than `base` and we denote by `val:digit→nat` the related natural injection. The numerals are represented by dependent lists of digits, and then the information about their length is carried by their type, at the static level. For all natural number `n` the type of length-`n` numerals is thus defined by:

`(num n)=(d_list digit n)`.

The function `Val:(∀n:nat)(num n)→nat` that associated with each numeral its value is the recursive function defined as follows:

`(Val 0 d_nil) = 0`
`(Val (n+1) (d_cons n d D)) = (val d)*base`^{n} `+ (Val n D)`

3.2 Specifying linear architectures

The specification of linear connections of 4-ported identical cells are parameterized by the types `A, B, C: Set` of the ports and by a relation

`cell:A→B→C→A→Prop`.

A connection is a relation over two length-`n` lists, an input carry and an output carry. More precisely, it is of type:

`connection: (∀n:nat)A→(d_list B n)→(d_list C n)→A→Prop`

and is defined inductively by the two propositions:

- `(∀a:A) (connection 0 a (d_nil A) (d_nil B) a)`
- `(∀n:nat) (∀a,a',a'':A) (∀b:B) (∀c:C)`
  `(∀lb:(list B n)) (∀lc:(list C n))`
  `(cell a b c a')→(connection n a' lb lc a'')→`
  `(connection (n+1) a (d_cons B n b lb) (d_cons C n c lc) a'')`

3.3 Example: the comparator

The comparator is a particular case of connection. The type `A` of the carry is a 3-valued inductive type `order:={L, G, E}`. The types `B` and `C` are instantiated by `digit`. For specifying the relation `cell` we first define a function `comparison: nat→nat→order` that associated with all natural numbers `v` and `v'` the value `L, G,` or `E` depending on whether `v` is less than `v'`, greater than `v'`, or equal to `v'`. The functional description of a cell is given by the term `f_cell: order→digit→digit→order` defined as follows:

```
(f_cell L d d')= L
(f_cell E d d')= (comparison (val d) (val d'))
(f_cell G d d')= G
```

The structure of the comparator is then specified by:

`(λn:nat)(λo:order)(λX,Y:(num n)) (connection n E X Y o).`

For describing its expected behavior, we introduce for all natural numbers `n` the type `[0,n[` with the natural injection `val_inf:(∀n:nat)[0,n[→nat`. The specification of the circuit is specified by the function `specif` of type

`(∀n:nat)[0,n[→[0,n[→order`

defined by:

```
(specif n x y):= (comparison (val_inf n x) (val_inf n y)).
```

The theorem of correctness which establishes that the implementation is correct with respect to the intended behavior can be informally stated as follows :

Let `n` *be a natural number and* `X` *and* `Y` *be two length-*`n` *numerals. Let us denote by* $\overline{\texttt{X}}$ *(respectively* $\overline{\texttt{Y}}$*) the value of type* `[0, base`^{n}`[` *of* `X` *(respectively* `Y`*). Then:*

`(∀o:order)(comparator n o X Y)` $\rightarrow$ `o=(specif base`^{n} $\overline{\texttt{X}}$ $\overline{\texttt{Y}}$`).`

However, because of the constant value of the input carry, the proof requires a generalization. Therefore a lemma is first established which sets forth the correct behavior of a connection, whatever value is given to the input carry. It is proven by induction on the term encoding the connection.

3.4 The factorization theorem

A more general approach oriented to the verification as well as to the synthesis of 1-dimension arithmetic circuits is given in [HDL90]. One can observe that, given a base `b`, each cell of the comparator implements a modulo-`b` version of the overall structure.

Let us characterize the relations that are implemented by such structures, and for that let us consider `R` a relation of type: `(∀n:nat)A→[0,n[→[0,n[→A→Prop`.

Proper relation. `R` *is said to be* `proper` *if and only if:*
`(∀n:nat)(∀a:A)(R 1 a` $\overline{0}$ $\overline{0}$ `a)`.

Moreover, let `n, m, x, x', q, q', r, r'` be natural numbers such that

`x=nq+r ; x'=nq'+r' ; x,x':[0, mn[ ; q,q':[0, m[ ; r,r':[0,n[`

The relation `R` is said to be `factorizable` if it holds on `x` and `x'` provided it holds on the quotients `q` and `q'` and on the remainders `r` and `r'`. More precisely, this notion is defined as follows.

Factorizable relation. `R` *is said to be* `factorizable` *if and only if:*

```
(∀a,a',a'':  A) (R m a q q' a') → (R n a' r r' a'') →
                                  (R mn a x x' a'').
```

The theorem of factorization states that for all relation `R` that is `proper` and `factorizable`, `(R b`^{n}`)` is implemented by a connection of `n` cells implementing `(R b)`. More formally:

```
(∀n: nat)(∀X, Y: (num n))(∀a, a':  A)(connection n a X Y a')→
                                     (R b^n a X̄ Ȳ a').
```

This theorem is proved by an induction on `(connection n a X Y a')`

3.5 Synthesis of linear structures

It is possible to use the Coq extraction mechanism to obtain a certified architecture that implements a behavior given by a proper and factorizable relation R. For that, the factorization theorem is set in a slightly different way. Let us denote by b the base of the system of numeration under consideration. First the relation R is specified by using a functional parameter FR of type (∀n:nat)A→[0,n[→[0,n[→A. Then, R is defined by:

```
(R n a x y a'):= a'=(FR n a x y).
```

The theorem is stated as follows:

$$(\forall n{:}\mathtt{nat})(\forall X,Y{:}\ (\mathtt{num}\ n))(\forall a{:}A)(\exists a'{:}A)(R\ b^n\ a\ \overline{X}\ \overline{Y}\ a').$$

The existential quantifier must be declared in the informative sort Set. The function extracted from such a proof will take as arguments a natural number n, two length-n numerals X and Y, and an element a:A. It will return an element a' of type A. The function f is certified to be such that $(R\ b^n\ a\ \overline{X}\ \overline{Y}\ a')$. The proof is performed by induction on n.

- If n=0, we give the witness a'=a and prove that $(R\ 1\ a\ \overline{0}\ \overline{0}\ a)$ using the fact that R is proper.
- Let us now consider a of type A, n a natural and two length-(n+1) numerals X=(d_cons n d D) and Y=(d_cons n d' D'). Let a_1 be (FR b a d d'). By induction hypothesis, there exists a' such that $(R\ b^n\ a_1\ \overline{D}\ \overline{D'}\ a')$. The relation being factorizable we can deduce that

  $$(R\ b^{(n+1)}\ a\ \overline{(\mathtt{d\_cons}\ n\ d\ D)}\ \overline{(\mathtt{d\_cons}\ n\ d'\ D')}\ a').$$

The term resulting from the extraction is in the system F_ω [Gir71]. From FR a function fr of type nat→A→nat→nat→A is obtained. Note that the logical contents of [0,n[have disappeared. The extracted function f is of type nat→list→list→A→A and defined by

- (f 0 D D' a)=a
- (f (n+1) (cons d D) (cons d' D') a)=(f n D D' a_1)
  where a_1=(fr b a d d')

From the extracted term, a ML program can be automatically generated. It produces the expected result, when taking as inputs a natural n and two length-n numerals X and Y. If one of the numerals is shorter than n, an exception is returned. Numerals longer than n are truncated.

The synthesis of the comparator is obtained by defining the function FR of type: (∀n:nat)order→[0,n[→[0,n[→order by:

```
(FR L x y)= L
(FR E x y)= (comparison (val_inf n x) (val_inf n y))
(FR G x y)= G
```

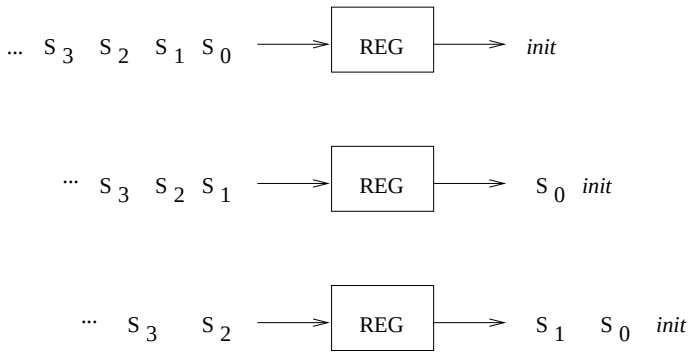

Figure 2: A register

and by applying the factorization theorem after having showed that the relation defined by

```
(R n a x y a'):= a'=(FR n a x y)
```

is `proper` and `factorizable`.

The rest of this paper is dedicated to a methodology for verifying sequential synchronous devices, based on a co-inductive axiomatization of automata, and to an application to a true circuit. Let us point out that, although we shall essentially emphasize the co-inductive aspect of the specifications, we shall keep using dependent lists to give accurate encoding of the component ports.

4 Co-inductive Axiomatization of Automaton Algebra

When we started thinking about a general axiomatization for synchronous sequential circuits, it appeared that it would basically rely on the way the history of the values carried by the wires would be represented. The purest and most elegant way to do this was to encode these infinite sequences as co-inductive streams as presented in section 2. So, no time parameters need handling and a register, for instance, is merely a function consing its initial value to its input stream (fig. 2).

The problem was then to avoid re-introducing any time parameter in the following. It was out of the question, for example, to represent expected behaviors by timing diagrams. Or to use functions that access the *nth* element of a given stream. Thus we were led to choose a representation by finite state machines that can be viewed as devices which compute an infinite sequence of outputs in response to an infinite sequence of inputs.

Mealy and Moore automata [Mea55, Moo56] have been broadly used for mod-

eling circuit structures. We found that this notion is also an adequate and elegant model for representing behaviors that are described by the designers in an informal way. As it will be illustrated in the case study we present in application, the description of the automata we handle is very abstract (for example a transition function may involve a complex algorithm over the natural numbers). This kind of representations, that are depicted by clear and compact transition diagrams, appears to be more trustworthy than other ones that are based on low level tools (like timing diagrams). In this uniform framework for encoding both the structures and the behaviors, these two notions are considered as descriptions of the same entity, represented at different abstraction levels. Moreover, the set of automata is equipped with composition rules [HU79, Boo67]. This algebraic structure leads naturally to modular descriptions of architectures and thus to the decomposition of complex device verification processes into the verification of simpler sub-components. Finally, we introduce a notion of equivalence over the automata which is a congruence for their composition rules. This makes a hierarchical approach possible since, in a modular device, a pre-proven sub-component can be replaced by its expected behavior.

In addition to the specific advantages of Mealy and Moore automata, and due to the expressiveness of the Calculus of Constructions, the following aspects of our encoding are worth being underlined:

– Genericity: Our definition of automata is generic enough to represent in a uniform way, both low level automata that are related to structures and more complex ones that represent behaviors.

– Compactness: Due to the high abstraction level of our axiomatization, we get extremely compact behavioral automata (at most 5 states in the example given as an application). The information is essentially carried by the state structure and by the transition and output functions.

– Co-inductive reasoning: Our co-inductive encoding leads to a single generic lemma that describes the proof schema that underlies any correctness proof. This lemma captures all the temporal aspects of the reasoning.

– Dependent types: As in the previous case study, we introduce dependent types whenever they contribute to a better precision of the specifications. They can be used jointly with co-inductive types to encode, for instance, a `n` boolean signal as a stream of length-`n` lists of booleans.

4.1 Specification of Mealy and Moore Automata

We present two variants of the notion of automaton, due to Mealy [Mea55] and Moore [Moo56]. The definitions are similar except that in a Mealy automaton, the output depends on the current state and on the input, whereas it only depends on the current state in a Moore automaton. It can be shown that the two notions are equivalent.

A Mealy automaton is defined by 5 parameters as follows.

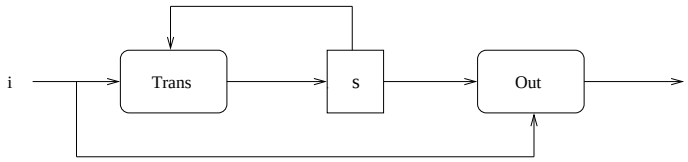

Figure 3: Representation of a Mealy automaton

Definition 1 *A Mealy automaton is a 5-uple* `(I, O, S, Trans, Out)` *where* `I`, `O` *and* `S` *are respectively the set of inputs, the set of outputs and the set of states.* `Trans` *is the transition function, of type* `I→S→S` *and* `Out` *is the output function, of type* `I→S→O` *(see fig. 3).*

Given an initial state `s`, the Mealy machine computes an infinite output sequence in response to an infinite input sequence `inp`. It can then be viewed as a function that we shall call *Mealy function*, of type `(Stream I)→S→(Stream O)` defined as a fixpoint as follows:

`fix.Mealy` λ`inp` λ`s (Cons (Out` $\texttt{inp}_0$ `s) (Mealy inp' (Trans` $\texttt{inp}_0$ `s))).`

The first element of the output stream is the result of the application of the output function `Out` to the first input $\texttt{inp}_0$ and to the initial state `s`. The tail of the output stream is then computed by a recursive call to `Mealy` on the tail `inp'` of the input stream and the new state `(Trans` $\texttt{inp}_0$ `s)`. This recursive call occurs just under the constructor `Cons` of the co-inductive type `Stream`, and that is the guarded condition to be met for the term to be well-formed.

This function `Mealy` depends in fact on the 5 parameters `(I, O, S, Trans, Out)` defining the automaton. As the first three parameters can be synthesized from the last two, we will denote it by `(Mealy Trans Out)`.

The stream of all the successive states from the initial one can be obtained similarly:

`(fix.States:(Stream I)→S→(Stream S))` λ`inp` λ`s (Cons s (States inp' (Trans` $\texttt{inp}_0$ `s))).`

One can notice that, with this encoding, the output stream can be defined without referring to the state stream, contrary to what happens when infinite sequences are defined as functions over the natural numbers where $(\texttt{o}_n)_{n:nat}$, the output sequence, and $(\texttt{s}_n)_{n:nat}$, the state sequence, are defined by :

`(`$\forall$`n:nat)` $\texttt{o}_n$`=(Out` $\texttt{inp}_n$ $\texttt{s}_n$`)` and $\texttt{s}_{n+1}$`=(Trans` $\texttt{inp}_{n+1}$ $\texttt{s}_n$`)`

Moore automata are described in fig. 4 and are defined as follows.

Definition 2 *A Moore automaton is a 5-uple* `(I, O, S, Trans, Out)` *where* `I`, `O` *and* `S` *are respectively the set of inputs, the set of outputs and the set of*

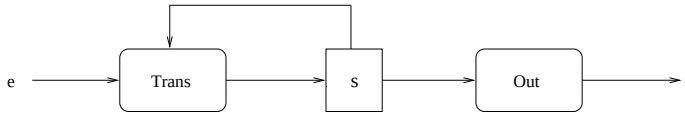

Figure 4: Representation of a Moore automaton

states. `Trans` *is the transition function, of type* `I→S→S` *and* `Out` *is the output function, of type* `S→O` *(see fig. 4).*

The Moore function associated with such an automaton is of type `:(Stream I)→S→(Stream O)` and is defined by:

`fix.Moore λinp λs (Cons (Out s) (Moore inp' (Trans inp`$_0$ `s))).`

It is well known, and we shall formally establish, that these two notions are equivalent in the sense that any Mealy machine can be simulated by a Moore machine and conversely.

Let us now axiomatize the inter-connection rules which express how a complex machine can be decomposed into simpler ones.

4.2 Modularity

Three basic inter-connection rules are defined on the set of automata [Boo67]. They represent the parallel composition, the sequential composition and the feedback composition of synchronous sequential devices.

4.2.1 Parallel Composition

In this paragraph we consider two Mealy automata respectively defined by the two following 5-uples:

`(I`$_1$`, O`$_1$`, S`$_1$`, Trans`$_1$`, Out`$_1$`)` and `(I`$_2$`, O`$_2$`, S`$_2$`, Trans`$_2$`, Out`$_2$`)`

and we define their Mealy functions

`A`$_1$ `:= (Mealy Trans`$_1$ `Out`$_1$`)` and `A`$_2$ `:= (Mealy Trans`$_2$ `Out`$_2$`).`

In the following and when it does not introduce ambiguity, `A`$_1$ and `A`$_2$ will be employed to denote both the automata and their Mealy functions.

The parallel composition of `A`$_1$ and `A`$_2$ is described in fig. 5. The two objects, on each side of the schema, need comments :

- `f=(f`$_1$`, f`$_2$`)` builds from the current input `i` the pair of inputs `(f`$_1$`(i), f`$_2$`(i))` for `A`$_1$ and `A`$_2$.
- `output` computes the global output from the outputs of `A`$_1$ and `A`$_2$.

More formally, the parallel composition of the two automata is defined by the following function over the input streams and initial states:

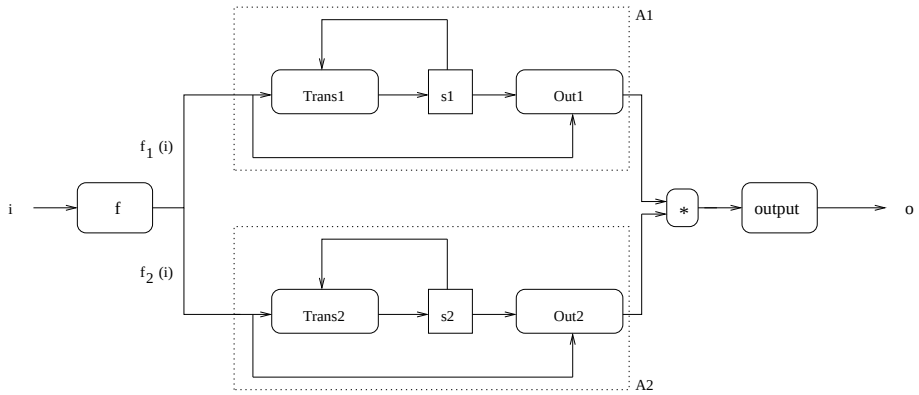

Figure 5: Parallel Composition

`parallel : (Stream I)`$\rightarrow$ `S1` $\rightarrow$ `S2 :=`
λ`inp` λ`s1` λ`s2 (Map output (Prod (A1 (Map f1 inp) s1) (A2 (Map f2 inp) s2))).`

In this definition, for `k` $\in \{1, 2\}$, `sk` is the initial state of `Ak` and `(Ak (Map fk inp) sk)` is the output stream of `Ak`. The global output stream is obtained by mapping the function `output` on the product of the output streams of `A1` and `A2`.

This parallel composition is not an automaton. But it can be shown that it is equivalent to a Mealy automaton called `PC` in the following sense: if a certain relation holds on the initial states, in response to the same input streams, the output streams for parallel composition and `PC` are equivalent. The type of its states is the product of the types of states of `A1` and `A2`. Its transition function and its output function are respectively defined by:

$\forall$`i:I)(`$\forall$`s1:S1)(`$\forall$`s2:S2)`
`(Trans_PC i (s1, s2))=((Trans1 (f1 i) s1), (Trans2 (f2 i) s2))`
`(Out_PC i (s1, s2))=(output (Out1 (f1 i) s1) (Out2 (f2 i) s2)).`

The following lemma states the equivalence between the parallel composition and this automaton.

Lemma 3 *Let* `PC` *be the Mealy function* `(Mealy Trans_PC Out_PC)`*. For all states* `s1` *of type* `S1`*,* `s2` *of type* `S2`*, and for all input stream* `inp` *of type* `(Stream I)` *the following relation holds:*

`(parallel inp s1 s2)` $\sim$ `(PC inp (s1,s2))`

Proof. Let us clarify on this simple example how a proof term can be built by co-inductive reasoning. The type of such a proof term being co-inductive, the term is itself a fixpoint. In the case of this lemma, it is particularly compact and defined as follows:

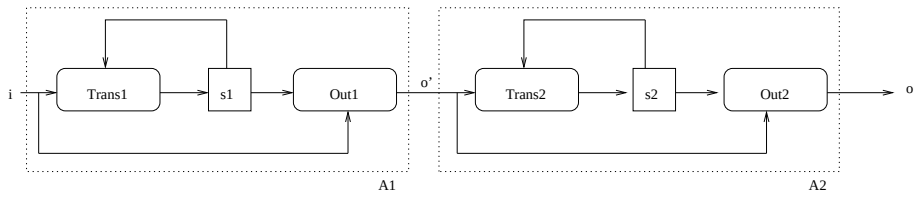

Figure 6: Sequential Composition of two Mealy Automata

```
fix.Equiv_parallel_PC λs1 λs2 λinp
                              (eqS h (Equiv_parallel_PC t1 t2 inp')).
```

Let us recall that `eqS` is the constructor of predicate $\sim$ (see section 2). The term `h` is a proof of the equality of the heads of the two streams under consideration, `inp'` is the tail of `inp` and t_1 and t_2 are the new states computed as follows:

t_1= `(Trans1 (f1 inp0) s1)` and t_2= `(Trans2 (f2 inp0) s2)`

Thus, the proof relies on the (co-inductive) hypothesis that the property being proved holds on the tail of the input stream and on the next states.

Such proofs are to be compared with those obtained when encoding infinite sequences by functions over the natural numbers. They are shorter and more elegant since they do not require handling indices nor performing induction.

The terms `parallel` and `PC` that have been defined in this paragraph depend in fact on several parameters. When these parameters are not clear from the context, we shall explicitly mention them. We shall write for example `(PC Trans1 Trans2 out1 out2 f output)` instead of `PC`. It is not mandatory to indicate the types of inputs, outputs and states, since they can be synthesized from the other parameters. This parameters management is handled by the section mechanism of the Coq proof assistant. Outside a parameterized section, the parameters are discharged and must appear explicitly in the terms that are defined in the section and that depend on the parameters.

4.2.2 Sequential Composition

Let us now consider two automata

`(I, O', S1, Trans1, Out1)` and `(O', O, S2, Trans2, Out2)`

the output set of the first one being the input set of the second one and let us define their Mealy functions:

`A1 := (Mealy Trans1 Out1)` and `A2 := (Mealy Trans2 Out2).`

The sequential composition is described in fig. 6 and merely corresponds to the composition of the functions `A1` and `A2`. As in the previous paragraph, we show that this composition is equivalent to an automaton `SC` the states of which are pairs composed of a state of `A1` and a state of `A2`. The transition and output

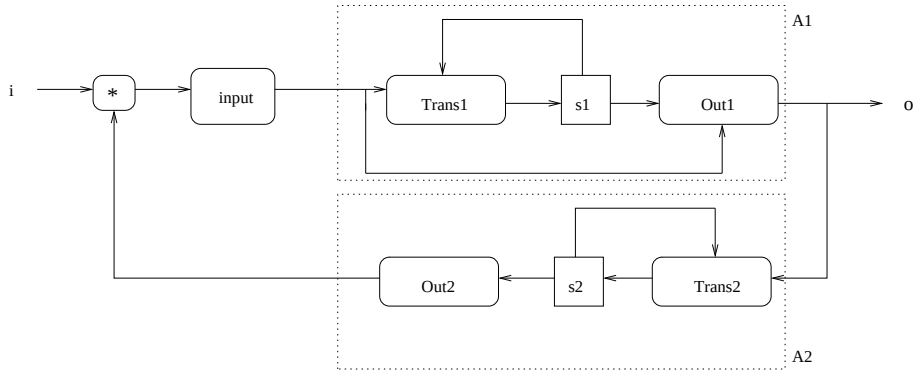

Figure 7: Feedback Composition

functions of SC are defined by:

$(\forall i{:}I)(\forall s_1{:}S_1)(\forall s_2{:}S_2)$
(Trans_SC i (s_1, s_2))= (($Trans_1$ i s_1), ($Trans_2$ (Out_1 i s_1) s_2))
(Out_SC (s_1, s_2))= (Out_2 (Out_1 i s_1) s_2))

More precisely, we have the following lemma:

Lemma 4 *Let* SC *be the Mealy function* (Mealy Trans_SC Out_SC)*. For all states* s_1 *of type* S_1*,* s_2 *of type* S_2*, and for all input stream* inp *of type* (Stream I) *the following relation holds:*

$$(A_2\ (A_1\ inp\ s_1)\ s_2) \sim (SC\ inp\ (s_1,s_2))$$

The proof is similar to that of the previous paragraph.

4.2.3 Feedback Composition

The previous two rules have been presented on Mealy automata but they could have been defined on Moore automata as well. For the feedback composition, at least one automaton must be a Moore automaton. Without this restriction, the specified interconnection is not correct since the current output depends on itself.

Let us consider a Mealy automaton and a Moore automaton, the output set of the first one being the input set of the second one. They are defined respectively by the 5-tuples:

$$(I_1,\ O,\ S_1,\ Trans_1,\ Out_1) \text{ and } (O,\ O_2,\ S_2,\ Trans_2,\ Out_2)$$

The corresponding Mealy and Moore functions are defined as usual by:

$$A_1 := (Mealy\ Trans_1\ Out_1) \text{ and } A_2 := (Moore\ Trans_2\ Out_2).$$

Let I be the set of the feedback composition inputs. The function input of type $I{*}O_2 \rightarrow I_1$ (where the operator * denotes the cartesian product) is a parameter

which computes the current input of A_1 from the current input i and the output of A_2. The following function out computes the current output of the device:

```
out:= λi λs1 λs2 (Out1 (input (i, (Out2 s2))) s1)
```

The functions updating the states are defined by:

```
upd1:= λi λs1 λs2 (Trans1 (input (i, (Out2 s2))) s1)
upd2:= λi λs1 λs2 (Trans2 (out i s1 s2) s2)
```

We can now specify the feedback function that computes the output stream produced by the device represented in fig. 7 with states having s_1 and s_2 as initial values, and in response to an input stream inp:

```
fix.feedback λinp λs1 λs2
           (Cons (out inp0 s1 s2) (feedback inp' (upd1 inp0 s1 s2)(upd2
inp0 s1 s2)))
```

As for the previous two rules, we show that this composition is equivalent to a Mealy automaton FC defined by the 5-uple (I,O, S_1*S_2, Trans_FC, Out_FC) where:

```
(Trans_FC i (s1, s2)) = ((upd1 i s1 s2), (upd2 i s1 s2)))
(Out_FC i (s1, s2)) = (out i s1 s2)
```

The following lemma establishes the equivalence of the two devices.

Lemma 5 *Let* FC *the Mealy function* (Mealy Trans_FC Out_FC)*. For all states* s_1 *of type* S_1 *and* s_2 *of type* S_2*, for all input stream* inp *of type* (Stream I)*, the following equivalence holds:*

$$\texttt{(feedback inp } s_1 \texttt{ } s_2\texttt{)} \sim \texttt{(FC inp (}s_1\texttt{, }s_2\texttt{))}$$

The proof is analogous to the previous ones.

These three composition rules are sufficient for describing all sequential circuit interconnections [Boo67].

4.3 Automaton Congruence

In this paragraph, we define a notion of congruence over the set of Mealy automata, which is useful in a hierarchical approach of circuit verification. It makes it possible to replace a pre-proven structural component by its equivalent behavioral automaton. Informally, we lift the stream equivalence to automata by stating that two automata are equivalent if their outputs are equivalent streams whenever their inputs are equivalent streams. It is then a kind of extensional equality over functions over streams.

Definition 6 *Let* (I, O, S_1, $Trans_1$, Out_1) *and* (I, O, S_2, $Trans_2$, Out_2) *be two automata and* A_1 *and* A_2 *their Mealy (or Moore) functions. The automata*

are said to be equivalent and we write: $\mathtt{A}_1 \equiv \mathtt{A}_2$ *if and only if:*

$$(\forall \mathtt{s}_1{:}\mathtt{S}_1)(\exists \mathtt{s}_2{:}\mathtt{S}_2)(\forall \mathtt{inp}{:}(\mathtt{Stream\ I}))(\mathtt{A}_1\ \mathtt{inp}\ \mathtt{s}_1) \sim (\mathtt{A}_2\ \mathtt{inp}\ \mathtt{s}_2) \wedge$$
$$(\forall \mathtt{s}_2{:}\mathtt{S}_2)(\exists \mathtt{s}_1{:}\mathtt{S}_1)(\forall \mathtt{inp}{:}(\mathtt{Stream\ I}))(\mathtt{A}_1\ \mathtt{inp}\ \mathtt{s}_1) \sim (\mathtt{A}_2\ \mathtt{inp}\ \mathtt{s}_2).$$

Lemma 7 *The condition* $\mathtt{A}_1 \equiv \mathtt{A}_2$ *is equivalent to:*

$$(\forall \mathtt{s}_1{:}\mathtt{S}_1)(\exists \mathtt{s}_2{:}\mathtt{S}_2)(\forall \mathtt{inp}_1,\mathtt{inp}_2 {:}(\mathtt{Stream\ I}))$$
$$(\mathtt{inp}_1 \sim \mathtt{inp}_2) \rightarrow (\mathtt{A}_1\ \mathtt{inp}_1\ \mathtt{s}_1) \sim (\mathtt{A}_2\ \mathtt{inp}_2\ \mathtt{s}_2) \wedge$$
$$(\forall \mathtt{s}_2{:}\mathtt{S}_2)(\exists \mathtt{s}_1{:}\mathtt{S}_1)(\forall \mathtt{inp}_1,\mathtt{inp}_2 {:}(\mathtt{Stream\ I}))$$
$$(\mathtt{inp}_1 \sim \mathtt{inp}_2) \rightarrow (\mathtt{A}_1\ \mathtt{inp}_1\ \mathtt{s}_1) \sim (\mathtt{A}_2\ \mathtt{inp}_2\ \mathtt{s}_2).$$

Proof. The proof consists in establishing first that for all automata `A`, for all states `s`, and for all equivalent input streams $\mathtt{inp}_1$ and $\mathtt{inp}_2$, $(\mathtt{A}\ \mathtt{inp}_1\ \mathtt{s}) \sim (\mathtt{A}\ \mathtt{inp}_2\ \mathtt{s})$. Then one uses the transitivity of the relation $\sim$.

Lemma 8 *The relation* $\equiv$ *is an equivalence relation over the set of automata.*

The proof is straightforward.

Lemma 9 *The relation* $\equiv$ *over the set of automata is a congruence for the three interconnection rules.*

That means that if $\mathtt{A}_1$, $\mathtt{B}_1$, $\mathtt{A}_2$, and $\mathtt{B}_2$ are automata, under the conditions over the input, output and state types for the interconnections to be correct, if $\mathtt{A}_1 \equiv \mathtt{B}_1$ and $\mathtt{A}_2 \equiv \mathtt{B}_2$ then $\mathtt{PC}_{A1,A2} \equiv \mathtt{PC}_{B1,B2}$, $\mathtt{SC}_{A1,A2} \equiv \mathtt{SC}_{B1,B2}$, and $\mathtt{FC}_{A1,A2} \equiv \mathtt{FC}_{B1,B2}$. Here $\mathtt{SC}_{A1,A2}$, $\mathtt{PC}_{A1,A2}$, and $\mathtt{FC}_{A1,A2}$ denote the Mealy automata resulting from the sequential, parallel and feedback composition of $\mathtt{A}_1$ and $\mathtt{A}_2$.

Finally, let us state precisely the equivalence between the Moore and Mealy definitions. In the two following lemmas we denote similarly the automata and their Moore or Mealy functions. It is trivial that for all Moore automata, there exists an equivalent Mealy automaton.

Lemma 10 *Let* `A = (I, O, S, Trans, Out)` *be a Moore automaton and let* `B` *the Mealy automaton defined by*

$$\mathtt{B = (I, O, S, Trans, (}\lambda\mathtt{s{:}S)(}\lambda\mathtt{i{:}I)(Out\ s))} \quad .$$

Then:

$$(\forall \mathtt{s{:}S})(\forall \mathtt{inp{:}(Stream\ I)})\ (\mathtt{A\ inp\ s}) \sim (\mathtt{B\ inp\ s})$$

In the converse implication, the two automata process on input streams one of them is the tail of the other.

Lemma 11 *For all Mealy automata, there exists an equivalent Moore automaton in the following sense. Let* `A = (I, O, S, Trans, Out)` *be a Mealy automaton. Let us define the functions:*

`Trans'` $:= (\lambda$`i:I)(`λ`(s,o):S*O)((Trans i s),(Out i s))` *and*
`Out'` $:= (\lambda$`(s,o):S*O)o`.

Let B *be the Moore automaton defined by* B = (I, O, S*O, Trans', Out'). *Then:*

(∀i:I)(∀inp:(Stream I))(∀s:S)
(A (Cons i inp) s) ∼ (B inp (Trans i s) (Out i s))

Proofs. The proofs of both lemmas are performed by co-induction.

4.4 Proof Schema for Circuit Correctness

Proving that a circuit is correct amounts to proving that, under certain conditions, the output stream of the structural automaton and that of the behavioral automaton are equivalent. We present in this section a generic lemma, all our correctness proofs rely on. It is in fact a kind of pre-established proof schema which handles the main temporal aspects of these proofs. Let us first introduce some specific notions.

In the following, we consider two Mealy automata :

$$A_1 = (I, O, S_1, Trans_1, Out_1) \text{ and } A_2 = (I, O, S_2, Trans_2, Out_2)$$

that have the same input set and the same output set.

Invariant. Given p streams, a relation which holds for all p-tuples of elements at the same rank is called an invariant for these p streams. For instance, if p=3, a predicate inv is defined co-inductively as follows.

Let I, S_1 *and* S_2 *be three sets,* P: I→S_1→S_2 →Prop *be a predicate, and* inp, st_1, st_2 *three variables of respective type* (Stream I), (Stream S_1), *and* (Stream S_2). *The predicate* P *is said to be invariant on the streams* inp, st_1, st_2 *and one writes* (inv P inp st_1 st_2) *if* P *holds on the heads of the streams and if* P *is invariant on the tails of the streams.*

Invariant state relation. *Let* R: S_1→S_2→Prop *and* P: I→S_1→S_2 →Prop *be two predicates.* R *is invariant under* P *for the automata* A_1 *and* A_2*, if and only if:*

(∀i:I)(∀s_1:S_1)(∀s_2:S_2) (P i s_1 s_2) → (R s_1 s_2) →
(R ($Trans_1$ i s_1) ($Trans_2$ i s_2)).

Output relation. *Let* R: S_1→S_2→Prop *be a relation over the states of two automata.* R *is an output relation if it is strong enough to induce the equality of the outputs, that is if and only if:*

(∀i:I)(∀s_1:S_1)(∀s_2:S_2) (R s_1 s_2) → (Out_1 i s_1) = (Out_2 i s_2).

We can now set forth the equivalence theorem:

Theorem 12 *Let* P: I→S_1→S_2→Prop *and* R: S_1→S_2→Prop *be two predicates. Let us assume that* R *is an output relation that is invariant under* P*, and*

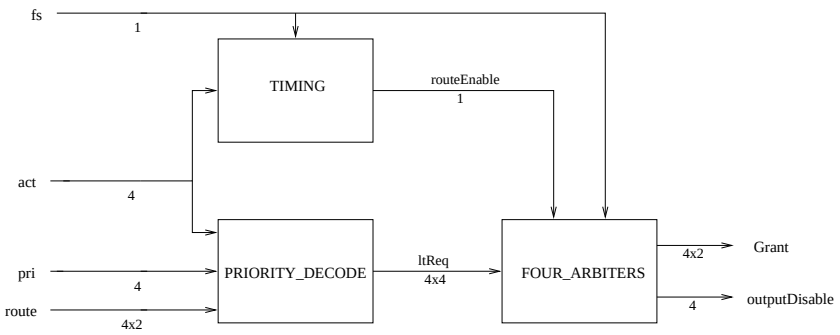

Figure 8: Arbitration Unit

that holds on two initial states $\mathtt{s}_1$ *and* $\mathtt{s}_2$*. Then, for all input streams* `inp`*:*

$$\mathtt{(Inv\ P\ inp\ (States\ Trans_1\ Out_1\ inp\ s_1)(States\ Trans_2\ Out_2\ inp\ s_2))} \\ \rightarrow \mathtt{(A_1\ inp\ s_1)} \sim \mathtt{(A_2\ inp\ s_2)}.$$

In other words if `R` is an output relation invariant under `P` that holds for the initial states, if `P` is an invariant for the common input stream and the state streams of each automata, then the two output streams are equivalent.

Proof. The proof of this lemma is done by co-induction.

This theorem will be systematically invoked when establishing the correctness of a circuit component as an equivalence between its structural automaton and its behavioral automaton.

5 Certification of a Realistic Circuit

In this section, we outline the certification process of a realistic circuit. It illustrates our methodology and gives evidence for its feasibility. The specifications rely not only on a co-inductive axiomatization of the history of the values carried by the wires but also on the use of dependent types for encoding precisely, and then reliably, the ports of the different components.

5.1 The 4 by 4 Switching Element

Designed and implemented at Cambridge University by the Systems Research Group, the Fairisle 4 by 4 Switch Fabric is an experimental local area network based on Asynchronous Transfer Mode (ATM).

The switching element is the heart of the 4 by 4 Switch Fabric, connecting 4 input ports to 4 output ports. Its main role is performing switching of data from input ports to output ports and arbitrating data clashes according to the output port requests made by the input ports. We focus here on its `ARBITRATION` unit, which is its most significant part, as far as specification and verification are concerned. This unit decodes requests from input ports and priorities between

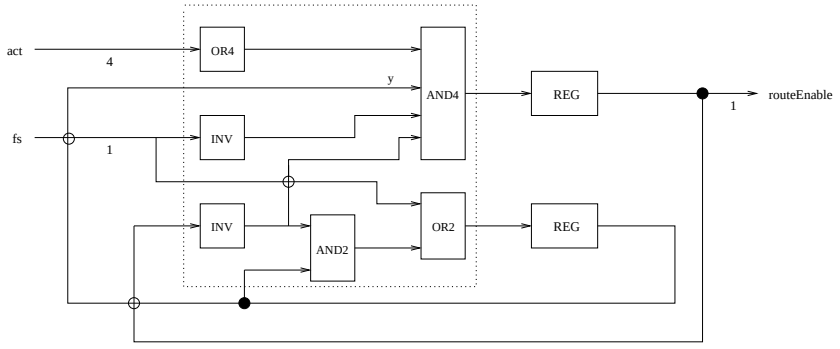

Figure 9: Timing Unit

data to be sent, and then it performs arbitration. It is the interconnection of 3 modules (fig.8) :

- `FOUR_ARBITERS` which performs the arbitration for all output ports, following the Round Robin algorithm,
- `TIMING` which determines when the arbitration process can be triggered,
- `PRIORITY_DECODE` which decodes the requests and filters them according to their priority. Its structure is essentially combinatorial.

As an illustration of section 4.4, we present in detail the verification of `TIMING`, which is rather simple and significant enough to illustrate the proof of equivalence between a structural automaton and a behavioral automaton. Then, we shall present how `ARBITRATION` is verified by joining together the various correctness results of its sub-modules. This will illustrate not only the hierarchical aspect of our approach, but also that the real objects we have to handle are in general much more complex than those presented on the example of `TIMING`.

5.2 Verification of the Unit Timing

The unit `TIMING` can be specified and proved directly, that is without any decomposition. It is essentially composed of a combinatorial part connected to two registers as shown in fig.9.

5.2.1 Structure

The structure of `TIMING` corresponds exactly to a Mealy automaton. Its transition function represents the boxed combinatorial part in fig. 9. The state is the pair of the two register values, encoded as a length-2 list of booleans. The unit input consists of a wire `fs` and a 4 wires signal `act` that is encoded by a length-4 dependent list of booleans. So, the parameters representing the input, output, and state types are instantiated as follows:

```
I := bool * (d_list bool 4)
O := bool
```

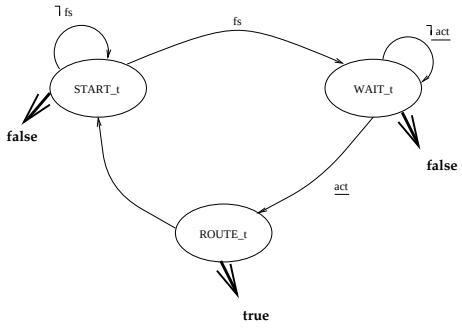

Figure 10: Timing Behavior

```
S := (d_list bool 2).
```

The functions `neg` and `andb` are the boolean negation and conjunction. The inverters and the four inputs *and* gates are defined by:

```
INV := neg.
AND_4 := (λa,b,c,d:bool) (andb a (andb b (andb c d))).
```

The automaton representing the circuit in fig. 9 is defined by means of its transition function `Trans_Timing` and its output function `Out_Timing` as follows:

```
Trans_Timing :  I→S→S := λ(fs, act) λ(s_1, s_2)
(List2 (AND_4 (OR_4 act) s_2 (INV fs) (INV s_1))
       (OR_2 fs (AND_2 (INV s_1) s_2)))

Out_Timing:I -> S -> O := λi λ(s_1, s_2) s_1

Structure_TIMING := (Mealy Trans_Timing Out_Timing).
```

In the definitions above, `List2` denotes the operator that builds a length-2 list from its two components.

5.2.2 Behavior

The expected behavior is presented in fig. 10. The output is a boolean value that indicates when the arbitration can be triggered. This output is false in general. When the frame start signal `fs` goes high, the device waits until one of the four values carried by `act` is true. In that case the output takes the value `true` during one time unit and then it goes low again. The type of the states is defined inductively as the set `S_beh`={`START_t, WAIT_t, ROUTE_t`}.

The transition function `trans_T` and the output function `out_T` are simple and defined by cases. The automaton `Behavior_TIMING` is obtained as usual by an instantiation of `Mealy`.

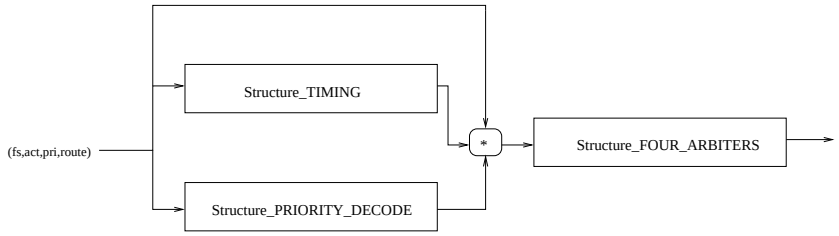

Figure 11: The Arbitration Structure as an Interconnection of Automata

5.2.3 Proof of equivalence

All the notions we use in this paragraph have been introduced in section 4.4. To prove the equivalence between `Behavior_TIMING` and `Structure_TIMING` we apply theorem 12. For that, we have to define a relation between the state (s_1, s_2) of the structure and the state `s` of the behavior. This relation, namely `R_Timing`, expresses that the first register value s_1 equals the output of the behavior in the state `s` (note that the value of this output does not depend on the current input), that insures that the relation is an *output relation*. It also expresses constraints between `s` and the second register s_2 of the structure.

$$\texttt{R\_Timing:S\_beh}\rightarrow\texttt{S}\rightarrow\texttt{Prop} := \lambda\texttt{s}\ \lambda(s_1, s_2)$$
$$((\forall \texttt{i:I})s_1=(\texttt{Out\_T i s})) \wedge (\texttt{s = START\_t} \wedge s_2 = \texttt{false} \vee$$
$$\texttt{s = WAIT\_t} \wedge s_2 = \texttt{true} \vee$$
$$\texttt{s = ROUTE\_t} \wedge s_2 = \texttt{true}).$$

No additional hypothesis is needed to prove that `R_Timing` is an invariant relation over the state streams, and this means that it is *invariant under* the `True` constant predicate over the streams, which is obviously an invariant.

The correction lemma is stated as follows:

Lemma 13 $(\forall \texttt{inp:(Stream I)})(\forall \texttt{s:S})(\forall \texttt{s\_beh:S\_beh})(\texttt{R\_Timing s\_beh s})\rightarrow$ $\texttt{(Behavior\_TIMING inp s\_beh)} \sim \texttt{(Structure\_TIMING inp s)}$.

In this lemma, as in all the correctness lemmas stating the equivalence between a structural automaton and a behavioral automaton, the combinatorial part of the proof and the temporal one are clearly separated. The former essentially consists in proving that `R_Timing` is an invariant under certain condition (`True` in this case), which is done by case analysis. The latter follows straightforwardly from theorem 12.

5.3 Hierarchical and Modular Aspects

Let us now illustrate the hierarchical modularity of our approach on the specification and the verification of an interconnection of several modules, namely the Arbitration unit (fig. 8). Its structure is described in fig. 11.

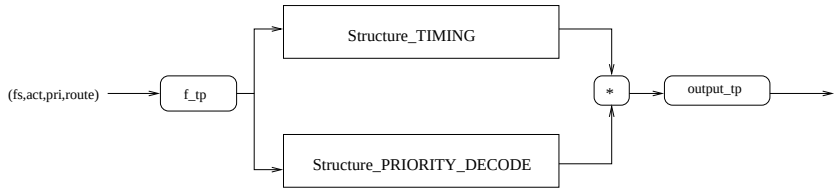

Figure 12: *Structure_TIMINGPDECODE*

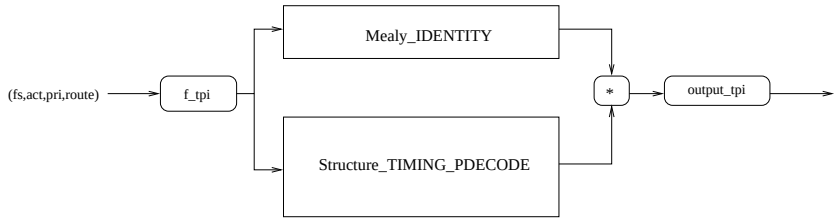

Figure 13: *Structure_TIMINGPDECODE_ID*

5.3.1 Structural Description

The structural description is given in several steps (fig. 12, 13, 14), by using the parallel and sequential composition rules (`PC` and `SC`) on automata presented in section 4.2.

For example, the final definition representing the structure of `ARBITRATION` as a sequential composition of two intermediate automata (fig. 14) is specified by the following definitions. The input type is:

```
I := bool*(d_list bool 4)*(d_list bool 4)*(d_list (d_list bool 2) 4).
```

and the circuit architecture is encoded by:

```
Structure_ARBITRATION : (Stream I)→S→(Stream O) :=
                        (SC Trans_TPD Trans_FA Out_TPD Out_FA).
```

where $\texttt{Trans}_{TPD}$ and $\texttt{Out}_{TPD}$ are the transition and output functions of the component `structure_TIMING- PDECODE_ID` and $\texttt{Trans}_{FA}$ and $\texttt{Out}_{FA}$ those of the component `structure_FOUR_ARBITERS`.

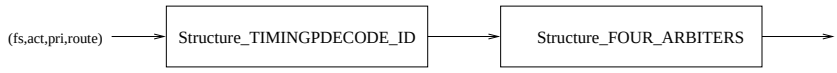

Figure 14: *Structure_ARBITRATION*

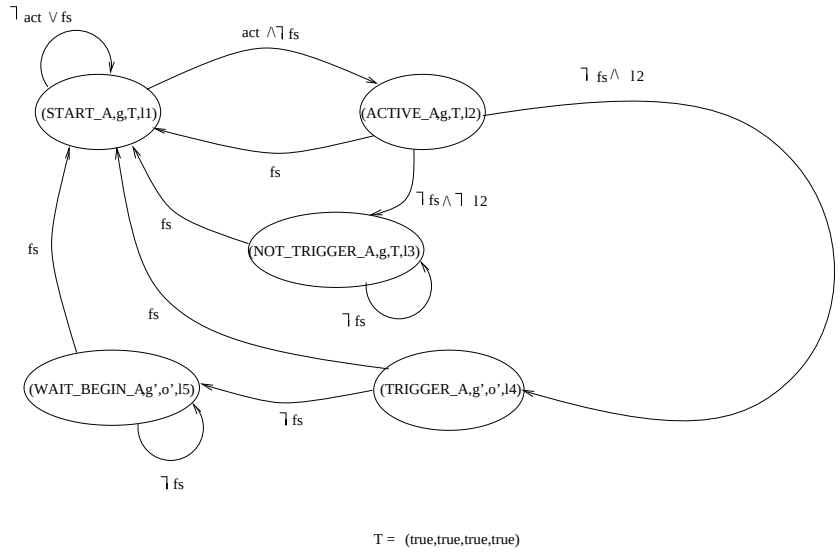

Figure 15: Arbitration Behavior

5.3.2 Behavioral Description

As for `TIMING`, the behavior of `ARBITRATION` was initially given in natural language by the designers. Several formal descriptions can be derived. For example, in HOL, Curzon uses classical timing diagrams (waveforms) whereas Tahar, in MDG, specifies it by means of Abstract State Machines (ASM) the states of which are located according to two temporal axis [TC96]. Our description is more abstract, more compact and closer to the designers intuition. We represent it, as usual, by a Mealy automaton which is described in fig. 15. It is particularly small (only 5 states). This comes from the fact that a great deal of information is expressed by the states themselves, the output function and the transition function.

The input has the same type `I` as in the structural description. The current input is then `(fs, act, pri, route)`. The states are 4-tuples consisting of:

- a label (`START_A` for instance),
- a list `g` of 4 pairs of booleans, each of them being the binary code of the last input port that gained access to the output port corresponding to its rank in the list `g`,
- a list `o` of 4 booleans indicating if the information of same rank in the list `g` above is up to date (an element of `g` can be out of date if the corresponding output port has not been requested at the previous cycle),
- the current requests `l`. It is a list of 4 elements (one for each output port). Each of these elements is itself a list of 4 booleans (one for each input port) indicating if the input port actually requests the corresponding output port and if it has a high level of priority or not.

The transition function computes the new state from the information carried by the current state and the current input. Hence, it is quite complex. Let us explain how `g` and `o` are updated. This is done by means of an arbitration process. Each output port first computes the last port (say `last`) that got access to this output port. Then, it makes a call to `RoundRobin(4, l, last)` where `RoundRobin` is described as follows :

```
RoundRobin(n, l, last) = if n=0 then
                           (last, true)
                         else
                          let succ = (last+1) mod 4 in
                           if succ ∈ l then
                              (succ, false)
                           else
                              RoundRobin (n-1, l, succ)
```

The list of the first components returned by the 4 calls (one for each output port) to `RoundRobin` constitutes the new value of `g` and the list of the second components, that of `o`. The new requests that are computed by decoding and filtering the current input.

We do not give the precise definition of the transition function which is rather complex. Let us just make clear that, at each step, it describes non trivial intermediate functions for arbitrating, filtering, decoding. This points out an essential feature of our specification: due to the high abstraction level of the CC specification language, we can handle automata which have few states but which carry a lot of information. This allows us to avoid combinatorial explosions and leads to short proofs (few cases have to be considered). We refer the interested reader to [Jak99] for the full details.

5.3.3 Proof of Correctness: an outline

The proof of correctness follows from the verification of the modules that compose the `Arbitration` unit. We perform it in several steps, hierarchically.

1. We build behavioral automata for `TIMING`, `FOUR_ARBITERS`, and `PRIORITY_DECODE`. We prove that these three automata are equivalent to the three corresponding structural automata.

2. We interconnect the structural automata and we get the global structural automaton called `Structure_ARBITRATION`.

3. In the same way, we interconnect the three behavioral automata in 1 and we get an automaton called `Composed_Behaviors`.

4. We show, from 1 and by applying the lemmas stating that the equivalence of automata is a congruence for the composition rules, that `Composed_Behaviors` and `Structure_ARBITRATION` are equivalent.

5. We prove that `Composed_Behaviors` is equivalent to the global expected behavior, namely `Behavior_ARBI- TRATION`. This is the essential part of the global proof and is much simpler than proving directly the equivalence between the structure and the behavior. As a matter of fact, `Composed_Behaviors` is more abstract than `Structure_ARBITRATION` which takes into account all the details of the implementation.

6. This final result, namely the equivalence of `Behavior_ARBITRATION` and `Structure_ARBITRATION`, is obtained easily from 4 and 5 by using the transitivity of the equivalence over the `Streams`.

Let us point out that theorem 12 has been applied several times (three times in 1 and once in 5).

Moreover, it is worth noticing that the `FOUR_ARBITERS` unit is itself composed of four sub-units and that its verification requires again a modular verification process.

6 Related Work

We do not pretend to give here an exhaustive bibliography of the many investigations in the field of hardware verification using theorem provers.

Let us mention the work closest to ours. In [PM95] Paulin-Mohring gives a proof of a multiplier, using a codification of streams in type theory, but she represents circuits as functions of time parameters and then she loses the benefit of handling streams. In [MJ96] streams are defined in PVS as an uninterpreted non empty data type constrained by axioms about uninterpreted constructor and accessors functions. This axiomatization is then used to verify a synchronous fault-tolerant circuit using co-inductive reasoning based on bisimulations. The Lava system developed at Chalmers University [Lav00] is based on a functional representation of circuits in Haskell. These circuits can then be simulated by an interpreter and verified by a prover system based on propositional logic. However, the verification of sequential devices is classically performed by induction over time parameters. Moreover the nature of the logic used in this system only allows to prove circuits of fixed size. More recently, in [BH01] the authors give a shallow embedding in Coq of a data-flow synchronous language. For that, they use co-inductive dependent types for encoding streams with *"absent elements"* and they apply the *"clock as types"* paradigm for expressing static synchronization constraints with a restricted form of dependent types.

The ATM Switch Fabric has been (and still is) widely used as a benchmark in the hardware community. For example, Schneider et al. [SK95] formally verified it using the verification system Mephisto which is based on the HOL theorem prover. Although they automated the verification of low-level submodules, they have not accomplished a complete verification. Other approaches on the Fabric propose abstraction processes in order to cope with the state explosion problem [LT98]. However, the authors cannot verify the whole fabric. In [GR96], Garcez et al. verify some properties of the fabric using the HSIS model checking

tool. But no model checking on the whole switch fabric model nor a verification against a high-level specification was reported. Curzon [Cur94a, Cur94b] has specified and proved this circuit using the HOL theorem prover. His study has been a helpful starting point for our investigations [JCC97] despite his approach is completely different in the sense that he specifies the structures as relations that are recursive on a time parameter and he represents the behaviors by timing diagrams. He does not obtain parameterized libraries but rather libraries related to specific pieces of hardware. As most of his proofs are inductive, each proof requires at least one particular induction, and sometimes several nested inductions with various base cases. This has to be contrasted with our unique generic temporal theorem. In [TSC+99] Tahar and al. proved the Fabric using MDG (Multiway Decision Graphs). They handle bigger automata and their proof is more automatic. However it is not reusable. Several comparisons between these various approaches can be found in [TCJ98, TC99].

7 Conclusion

In this paper, we have thoroughly investigated how to take advantage of the expressiveness of the Calculus of (Co)-Inductive Constructions in the field of hardware verification. After a short case study on the use of dependent types and extraction for specifying and synthesizing arithmetic linear structures, we have presented a high level generic methodology, entirely implemented in Coq, for modeling and verifying synchronous sequential circuits. The starting point of this approach is a uniform co-inductive description of the structures and the expected behaviors of circuits by means of Mealy automata. Then we have demonstrated its applicability to a realistic circuit.

The points of our work that must be emphasized are the following:

1. *The use of extraction for synthesizing a class of circuits* We have used the constructive aspect of the logic to synthesize a class of circuits that implement proper and factorizable arithmetic relations. The Coq extraction mechanism allowed us to get a certified description of these devices from a proof term of an existential lemma.

2. *The use of dependent types for reliable specifications.* Although the benefit of dependent types for hardware specification has often been pointed out by several authors([HDL90, Lee92] ...), few significant developments using them have actually been achieved, since they are known to be rather tricky to use. Curzon reported that some errors in its specifications caused a big lost of time in his verification process. An early detection of these errors (at the specification stage) would have been possible by using dependent types. Thus, we have introduced dependent types whenever they contributed to a better reliability of the specifications and we have shown that an heavy use of this kind of encoding was feasible in practice. Several reusable axiomatizations have been done (for handling lists, numeration systems, repetitive arithmetic structures) in which generic properties and theorems have been proven.

3. *The use of co-inductive types for neat specifications.* We could obtain a clear and natural modeling of the history of the wires in a circuit as well as the automata behaviors without introducing any temporal parameter.

4. *Co-inductive reasoning for elegant and clear proofs processes.* We could capture once and for all in one generic theorem most of the temporal aspects of the proofs. In each specific case, only combinatorial parts need to be developed. So, the proof process is clarified, simplified and is made more generic.

5. *High abstraction level in axiomatizing automata.* Our definition of automata is generic enough to represent in a uniform way, both low level automata that are related to the structures and more complex ones that represent behaviors. Due to the high abstraction level of this axiomatization, we get extremely compact behavioral automata (at most 5 states in the example given as an application). This comes from the complex structure of the states that carry a lot of information. Therefore the proofs by cases are short but they make use of high level transition functions on rich data types.

6. *The feasibility of our approach*, that has been demonstrated on the example of a realistic non trivial circuit. Our whole development (including the generic tools and the case study on the linear arithmetic structures) takes approximatively 13,000 lines.

7. *The hierarchical and modular approach.* Not only does this lead to clearer and easier proof processes but also it allows us to use, in a complex verification process, correctness results related to pre-proven components.

This work results in several reusable Coq libraries.

References

[BH01] Sylvain Boulmé and Grégoire Hamon. Certifying synchrony for free. In *Logic for Programming, Aritificial Intelligence and Reasoning, LPAR 2001*, Havana, Cuba, December 2001.

[Boo67] Taylor L. Booth. *Sequential machines and automata theory.* John Wiley and Sons, Inc., 1967.

[Bry86] Randal Bryant. Graph-based algorithms for boolean function manipulations. *IEEE Transactions on Computers*, C-35(8), 1986.

[CGJ96] Solange Coupet-Grimal and Line Jakubiec. Coq and hardware verification : a case study. In *The 1996 International Conference on Theorem Proving in Higher Order Logics, TPHOL'96*, Turku, Finland, August 1996. Springer-Verlag.

[CH85] Thierry Coquand and Gérard Huet. Constructions : A Higher Order Proof System for Mechanizing Mathematics. *EUROCAL 85, Linz Springer-Verlag LNCS 203*, 1985.

[CJ99] Solange Coupet-Grimal and Line Jakubiec. Hardware verification using co-induction in coq. In *TPHOLs'99*, LNCS 1690, Nice, France, September 1999.

[Cur94a] Paul Curzon. Experiences Formally Verifying a Network Component. In *Proceedings of the 9th Annual IEEE Conference on Computer Assurance*, pages 183–193. IEEE Press, 1994. (Time taken, problems encountered, errors found for the 4 by 4 fabric verification).

[Cur94b] Paul Curzon. The Formal Verification of the Fairisle ATM Switching Element. Technical Report 329, University of Cambridge, March 1994.

[Gim96] Eduardo Giménez. *Un Calcul de Constructions Infinies et son application a la vérification de systèmes communicants*. Thèse d'université, Ecole Normale Supérieure de Lyon, December 1996.

[Gir71] Jean-Yves Girard. Une extension de l'interprétation de gödel l'analyse et son application l'élimination des coupures dans l'analyse et la théorie des types. *Studies in —Logic and the Foundations of Mathematics*, 63, 1971.

[GR96] E. Garcez and W. Rosenstiel. The Verification of an ATM Switching Fabric using the HSIS Tool. *IX Brazilian Symposium on the Design of Integrated Circuits*, 1996.

[HDL90] F.K. Hanna, N Daeche, and M Longley. Specification and Verification Using Dependent Types. *IEEE Transactions on Software Engineering*, 16(9):949–964, September 1990.

[HU79] L. Hopcroft and L. Ullman. *Introduction to automata theory, languages and computation*. Addison-Wesley Publishing Company, 1979.

[Jak99] Line Jakubiec. *Vérification de Circuits dans Coq*. Thèse d'université, Université de Provence, Juillet 1999.

[JCC97] Line Jakubiec, Solange Coupet-Grimal, and Paul Curzon. A Comparison of the Coq and HOL Proof Systems for Specifying Hardware. In Elsa L. Gunter and Amy Felty, editors, *Supplementary Proceeding of TPHOLs'97*, LNCS, pages 63–78, Murray Hill, NJ, USA, 19-22th August 1997.

[Lav00] The Lava Homepage. http://www.cs.chalmers.se/∼koen/Lava, 2000.

[Lee92] Miriam Leeser. Using Nuprl for the Verification and Synthesis of Hardware. In C. A. R. Hoare and M. J. C. Gordon, editors, *Mechanized Reasoning and Hardware Design*, International Series on Computer Science, pages 49–68. Prentice Hall, 1992.

[LM90] I.M. Leslie and D.R. McAuley. Fairisle : A General Topology ATM LAN. *http://www.cl.cam.ac.uk/ Research/SRG/fairpap.html*, December 1990.

[LM91] I.M. Leslie and D.R. McAuley. Fairisle : An ATM Network for the Local Area. *ACM Communication Review*, 4(19):327–336, September 1991.

[LT98] J. Lu and S. Tahar. Practical Approaches to the Automatic Verification of an ATM Switch using VIS. In *IEEE 8th Great Lakes Symposium on VLSI (GLS-VLSI'98)*, pages 368–373, Lafayette, Louisiana, USA, February 1998. IEEE Computer Society Press.

[Mea55] George H. Mealy. A Method for Synthesizing Sequential Circuits. In *Bell System Technical Journal*, volume 34(5), pages 1045–1079, 1955.

[MJ96] Paul S. Miner and Steven D. Johnson. Verification of an Optimized Fault-Tolerant Clock Synchronization Circuit. In *Designing Correct Circuits*. Båstad, 1996.

[Moo56] Edward F. Moore. Gedanken-experiments on Sequential Machines. In C.E. Shannon and J. McCarthy, editors, *Automata studies*, pages 129–153. Princeton University Press, 1956.

[PM95] Christine Paulin-Mohring. Circuits as Streams in Coq. Verification of a Sequential Multiplier. *Basic Research Action "Types"*, Juillet 1995.

[PM96] Christine Paulin-Mohring. *Définitions Inductives en Théorie des Types d'Ordre Supérieur*. Habilitation à diriger les recherches, Université Claude Bernard Lyon I, December 1996.

[SK95] K. Schneider and T. Kropf. Verifying Hardware Correctness by Combining Theorem Proving and Model Checking. In J. Alves-Foss, editor, *International Workshop on Higher Order Logic Theorem Proving and Its Applications: B-Track: Short Presentation*, pages 89–104, August 1995.

[TC96] Sofiène Tahar and Paul Curzon. A Comparison of MDG and HOL for Hardware Verification. In J. Grundy J. Von Wright and J. Harrison, editors, *TPHOLs'96*, LCNS 1125, pages 415–430, Turku (Finlande), 27-30th August 1996. Springer-Verlag.

[TC99] Sofiène Tahar and Paul Curzon. Comparing HOL and MDG: a Case Study on the Verification of an ATM Switch Fabric. *Nordic Journal of Computing*, 6(4):372–402, 1999.

[TCJ98] S. Tahar, P. Curzon, and Lu J. Three Approaches to Hardware Verification : HOL, MDG and VIS Compared. In G. Gopalakrishnan and P. Windley, editors, *Formal Methods in Computer-Aided Design*, LCNS 1522, pages 433–450, FMCAD'98, Palo Alto, California, USA, November 1998. Springer-Verlag.

[Tea01] The Coq Development Team. The Coq Proof Assistant Reference Manual – Version V7.1. Technical report, LogiCal Project-INRIA, 2001.

[TSC+99] Sofiène Tahar, Xiaoyu Song, Eduard Cerny, Michel Langevin, and Otmane Ait-Mohamed. Modeling and Verification of the Fairisle ATM Switch Fabric using MDGs. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 18(7):956–972, 1999.