Under consideration for publication in Formal Aspects of Computing

Certifying Circuits in Type Theory

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

Laboratoire d'Informatique Fondamentale de Marseille (UMR 6166)
[1]CMI - Université de Provence, 39 rue Joliot-Curie, F-13453, Marseille, France.
Solange.Coupet@lif.univ-mrs.fr;
[2]Faculté des Sciences de Luminy - Université de la Méditerranée, 163 Avenue de Luminy 13288 - Marseille Cedex-9.
Line.Jakubiec@lif.univ-mrs.fr

Abstract. 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 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.

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 have 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 Inductive Constructions (CC). The motivation for choosing this logical framework is threefold. First, its expressiveness makes it possible to give clear, accurate and generic specifications, to reason elegantly about 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.

Correspondence and offprint requests to: Solange Coupet-Grimal and Line Jakubiec

The first part of this work deals with combinatorial circuits and their representation with dependent types. It has often been argued that dependent types are very valuable for specifying hardware precisely and reliably. As one can lose a lot of time in reasoning about invalid specifications, it is important to be able to detect errors early, when type-checking the specifications. This is made possible by using dependent types. However CC dependent types are known to be rather tricky to use in practice. One of our contribution is to provide several Coq reusable axiomatizations (for handling lists, numeration systems, repetitive arithmetic structures) in which generic properties and theorems have been proven. Moreover, when a lot of information is carried within the types, problems such type incompatibility can occur later on, during the proof process. So, another point of interest of our work is a feasibility study on practical examples. We start by a small sized case study based on [HDL90]. Its significance comes from the heavy use of various kinds of dependent types. We also experiment on this example with the Coq extraction mechanism for synthesizing a class of circuits [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 lemma about automata equivalence from which follow all the correctness proofs in the practical cases. This lemma 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 apply this methodology to a real non trivial device, 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 compact representations with high level transition functions on rich data types [CJ99].

This paper is organized as follows. Section 2 is a brief introduction to the Calculus of Inductive Constructions. Section 3 deals with combinatorial circuits, dependent types, and extraction. Section 4 is dedicated to axiomatizing automata. Then, in section 5, we demonstrate the feasibility of our approach on 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 Inductive Constructions

The Calculus of Inductive Constructions (CC) 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 Coq proof assistant [Tea01], a tactic oriented proof-checker. In Coq, developments can be split into various parameterized modules to be separately verified. Thus, several developments can share modules that, being compiled once and for all, are loaded quickly. Let us now review the main features of CC (for readability, we do not use in this paper the exact Coq syntax).

The Curry-Howard isomorphism

CC provides in a uniform logical framework both a functional programming language for specifications and a higher order intuitionistic logic for reasoning about specifications. Indeed, its logic relies on the *propositions-*

as-types correspondence: a proposition is a type and a proof is a term inhabiting this type, so that proving amounts to type checking. In fact, 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`. For this reason, types are classified in two twin sorts, `Prop` and `Set`, corresponding to their two possible semantics.

Inductive and Co-inductive types

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. Inductive types represent least fixpoints, whereas co-inductive types represent greatest fixpoints. 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.
CoInductive c_list : Set := c_nil : c_list | c_cons : A→c_list→c_list.
CoInductive Stream : Set := Cons : A→Stream→Stream.
```

All these types denote sets of lists the elements of which are in `A`. Inductive type `list` is the set of the well-founded (finite) lists. Co-inductive type `Stream` is the set of the infinite lists. Co-inductive type `c_list` is the set of all the finite or infinite lists. Let us mention that these types depend on parameter `A`, that is *discharged* outside the current Coq section. For instance, outside the section, the type `Stream` is polymorphic and the type of the streams of elements in `A` is `(Stream A)`.

Inductive reasoning

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. Here is the induction principle for the type `list`:

$$(\forall \mathtt{P} : \mathtt{list} \to \mathtt{Prop})\ ((\mathtt{P}\ \mathtt{nil}) \to ((\forall \mathtt{a} : \mathtt{A})(\forall \mathtt{l} : \mathtt{list})(\mathtt{P}\ \mathtt{l}) \to (\mathtt{P}\ (\mathtt{cons}\ \mathtt{a}\ \mathtt{l}))) \to (\forall \mathtt{l} : \mathtt{list})(\mathtt{P}\ \mathtt{l})).$$

Total functions can be defined by structural recursion on terms the type of which is inductive. For example, the `length` of finite lists is defined by:

$$\mathtt{fix\ length}.\ (\lambda \mathtt{l} : \mathtt{list})\ (\mathtt{cases}\ \mathtt{l}\ \mathtt{of}\ \mathtt{nil}\ \Rightarrow\ \mathtt{0} \mid (\mathtt{cons}\ \_\ \mathtt{l}')\ \Rightarrow\ \mathtt{1} + (\mathtt{length}\ \mathtt{l}')).$$

Co-inductive reasoning

Recursive functional terms ranging over a co-inductive type `T` can be interpreted

- either (if `T` is in sort `Set`) as lazy recursive programs that compute infinite objects,
- or (if `T` is in sort `Prop`) as proofs using as hypothesis the proposition `T` itself: this corresponds to a recursive call in the proof term.

Such a term must satisfy a syntactic *guard condition* to be well-formed: the recursive calls must occur *under* a constructor of `T`. This condition is dual to the fact that, when defining functions by structural induction on a well-founded term, the recursive calls must apply to strict subterms of the initial argument.

In the rest of this paper we shall write σ_0 for the head of a stream σ and σ' for its tail. Thus, the mapping of a function `f` on a stream σ is defined as follows (for simplicity, we do not mention the types):

$$\mathtt{fix\ Map}.\ \lambda \mathtt{f}\ \lambda\sigma\ (\mathtt{Cons}\ (\mathtt{f}\ \sigma_0)\ (\mathtt{Map}\ \mathtt{f}\ \sigma')).$$

This term is well-formed since the recursive call to `Map` is guarded by the constructor `Cons`. We describe a co-inductive proof in detail in section 4.3.2.

Our methodology for verifying synchronous sequential circuits relies on encoding by streams the history of the values carried by the wires. 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
```

Dependent types

In CC, types can depend on terms. For example, one can define the type of *dependent lists* as follows:

```
Inductive d_list:nat->Set:=
     d_nil:(d_list 0) | d_cons:(n:nat)A→(d_list n)→(d_list (n+1))
```

For all `n` of type `nat`, `(d_list n)` is a type that depends on term `n`. It is interpreted as the sets of length-`n` lists. 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.

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 (whose type is in sort `Prop`), exactly as a compiler ignores comments. In such a way, one obtains the purely computational component `f` (whose type is in sort `Set`) which has been proved to be correct. This is how the system Coq can provide a mechanism which automatically extracts from a term the certified computational part.

The following section handles combinatorial circuits. It exemplifies the use of dependent lists for specifying connections and the use of the Coq extraction mechanism for 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. Combinatorial Aspects and Dependent Types

We focus in this section on the practicality of CC dependent types for describing some combinatorial aspects of circuits. CC dependent types are known to be difficult to use in practice. This comes from the fact that there is no rule of the form:

$$\frac{p : A(t) \quad q : (t{=}s)}{p : A(s)}$$

Here the equality under consideration is Leibniz equality, which is in general undecidable. Thus, such a rule would make the type checking undecidable. Therefore, it is impossible to encode in the calculus some quite natural statements. For example, it is impossible to express that a dependent length-`n` list is the empty list `d_nil` whenever `n` equals 0. Indeed, if `l` is a term of type `(d_list n)`, even under the hypothesis `n = 0`, the term `l = d_nil` is not typable since:

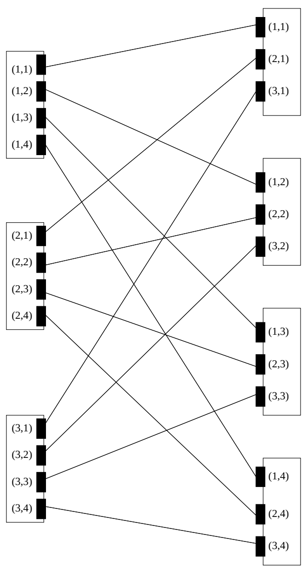

Fig. 1. A transposed connection

– `l` is of type `(d_list n)`,
– `d_nil` is of type `(d_list 0)`,
– such an equality, to be typable, requires the two terms to have the same type.

Very often, nice ideas on integrating much information into types must be given up due to this kind of problems that can also occur later on, during the proof process. However, the benefit of dependent types for hardware specifying is well known (see for example [HDL90, Lee92]). As CC provides many valuable features and is implemented in a very reliable system, it appeared to us that it was of interest to investigate whether or not dependent types can be handled significantly to specifying circuits. We answer positively this question, and in order to simplify the work of potential hardware verifiers with Coq, we provide substantial general libraries. Let us give some examples.

3.1. Ports and Connections

Contrary to model checking methods, interactive verification using a proof assistant allows us to deal with circuits with parameterized size. Such circuits can be modeled using types that depend on this parameter. In that way, specification errors can be detected early by the type checker, before starting the proof process. Here is an example that illustrates how a systematic use of dependent lists can circumvent such errors.

The ports of a circuit composed of `n` identical `m`-ports modules, can be specified by a length-`n` list of length-`m` lists of boolean values. Then, connections between such devices are described by functions on dependent lists. Figure 1 depicts the particular case of a *tranposed connection* between a circuit with 3 outputs of 4 booleans and a circuit with 4 inputs of 3 booleans. Such a connection amounts in fact to compute the transposed matrix of a 3×4 matrix. More generally, it is specified by a function `transpose`, of type:

```
(∀n,m:nat)(d_list (d_list Bool n) m) →(d_list (d_list Bool m) n).
```

This function is defined by recursion on `n` and uses two functions that take in argument a list of lists and compute the list of their heads and the list of their tails.

Such an approach leads to performing proofs at specification time. This is illustrated by the next example. It is quite technical and is not essential for the rest of the paper.

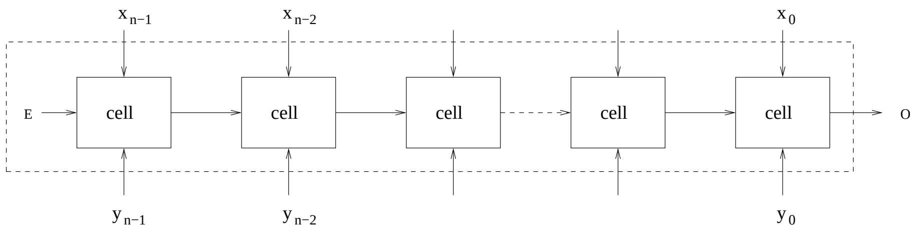

Fig. 2. A linear arithmetic structure

Example Let `A` be a set on which Leibniz equality is decidable. One can define inductively a predicate `member` that expresses that an element `a` of type `A` occurs in a dependent list `l`:

```
Inductive member: A→(∀n:nat)(d_list n)→Set:=
    eq_hd : (∀n:nat)(∀l:(d_list n)) (member a (n+1) (d_cons n a l))|
    in_tl: (∀b:A)(∀n:nat)(∀l:(d_list n)) (member a n l) → (member a (n+1) (d_cons n b l)).
```

We can now define a function `remove` that removes the first occurrence in a list `l` of an element that is member of `l`. Such a function takes as arguments not only the element `a` to be removed, a natural number `n`, and a length-`(n+1)` list `l`, but also a proof `p` that `a` is in `l`. Therefore, such a function as type `(∀a:A)(∀n:nat)(∀l: (d_list (n+1))) (member a (n+1) l)→(d_list n)`. It has the following form:

```
fix remove.λa:A λn:nat
Cases n of
         O     => λl: (d_list 1) λp:(member a 1 l) d_nil|
         (m+1) => λl: (d_list (n+1)) λp:(member a (n+1) l)
                  if a=(head n l) then (tail (n+1) l)
                  else (d_cons m
                               (head n l)
                               (remove a m (tail (n+1) l) q)).
```

In the recursive call above, term `q` is a proof that `a` is in the tail of `l`. It is built from the proof of a previously established lemma that states that *if an element of a list* `l` *is not equal to its head, then it is member of its tail.*

We provide a 1000 lines library dedicated to such functions on dependent lists [CJa, CJb].

3.2. Numeration Systems and Regular Architectures

In order to verify a certain class of arithmetic circuits, one has to handle numeration systems. We have axiomatized such notions, making use of various kind of dependent types. In addition to dependent lists, we use subsets of type {`x:A | (P x)`}. A term inhabiting such a type is a pair made of an element `a` of `A` and a proof `p` of `(P a)`. A numeration system is defined by a base `b` of type {`x:nat | x>0`}. In this system a digit is a term of type {`x:nat | x < b`}, and a *numeral* `N` is a length-`n` list of digits. Such a numeral `N` denotes a natural number `v` that we define by structural recursion on `N`, and we prove that `v <b`^{n}. Thus one can associate with any numeral of type `(d_list digit n)` its value `V` of type {`x:nat | x<b`^{n}}.

This serves as a basis for an experimentation on the specification and the verification of a class of linear arithmetic architectures as suggested in [HD92]. These combinatorial circuits (Fig. 2) are connections of identical cells with 4 ports : a carry-in, a carry-out, and two inputs x_i et y_i. Globally, such a connection can be viewed as a device with a carry-in, a carry-out, and two inputs that are the numerals constituted of the digits x_i and y_i. It is specified by an inductive relation `Connection` of type:

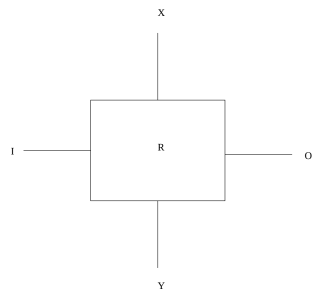

Fig. 3. Relation R

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

We prove a theorem of factorization on the arithmetic relations susceptible of being implemented by this kind of linear devices. Let us consider relations `R` (Fig. 3) of type :

```
R :(∀b:nat) A → {x:nat | x<b} → {x:nat | x<b} →A →Prop
```

Given a natural number `b`, such relations apply to an element `I` in set `A`, two natural numbers `X` and `Y` less than `b`, and an element `O` in `A`.

Informally, such a relation is said to be *proper* if the two arguments `I` and `O` are equal whenever `X` and `Y` are null. `R` is said to be *factorizable* if it holds on `X` and Y whenever it holds for the quotients and the remainders of the euclidean divisions of `X` and `Y` by a natural number `n` (with adequate values for `I` and `O`).

The factorization theorem states that provided the relation R is *proper* and *factorizable*, (R b^n) is implemented by a `Connection` of `n` cells that implement `(R b)`. By this theorem, one can verify the whole class of linear architectures that realize such proper and factorizable arithmetic relations (comparison, addition, subtraction, multiplication by a constant, division by a constant, remainder modulo a constant).

Let us mention that we also have experimented Coq extraction on this example. For that, the theorem of factorization is stated existentially. More precisely, assuming that `R` is proper and factorizable, one proves that for all `a` in `A`, for all natural numbers `n`, and for all length- `n` numerals `X` and `Y`:

$(\exists\ a':A)\ (R\ b^n\ a\ V_X\ V_Y\ a')$

where V_X and V_Y are the values of `X` and `Y` in type $\{x:nat \mid x<b^n\}$. As the underlying logic is constructive, the proof term contains an algorithm that constructs from an input `a` a *witness* `a'` that satisfies $(R\ b^n\ a\ V_X\ V_Y\ a')$. Therefore, a certified functional description of an architecture that implements $(R\ b^n)$ can be extracted from the proof. This is a step towards circuits synthesis, but it remains work to obtain concrete descriptions (in VHDL for example) from such high level descriptions.

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 encodings of the component ports.

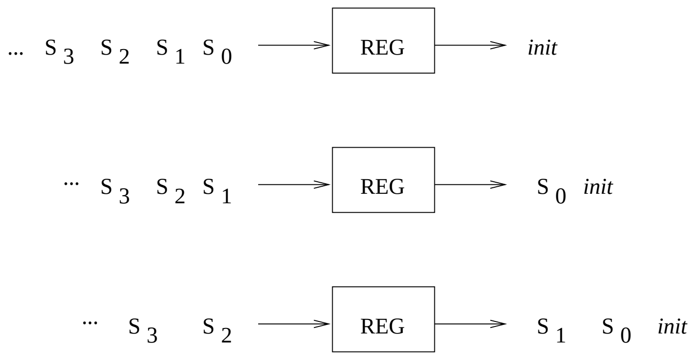

Fig. 4. A register

4. Sequential Circuits and Co-inductive Axiomatization of Automaton Algebra

We propose an approach to specify and verify sequential circuits, suggested by the expressiveness of CC. It is founded in particular on *higher-order co-inductive* representations, also involving *dependent types.*

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. We adopt an algebraic approach, in which these infinite sequences are encoded as co-inductive streams (cf. section 2). This leads to neat specifications in which no time parameters need handling. A register, for instance, is merely a function *consing* its initial value to its input stream (fig. 4).

The problem was then to avoid re-introducing any time parameter in the following. It was out of the question, for example, to use functions that access the $\mathtt{t}^{th}$ element of a given stream or to represent expected behaviors by timing diagrams. Indeed, using such descriptions leads to working with *temporal* statements such as: "*at all time points* $\mathtt{t}$ *comprised between the initial instant* $\mathtt{t}_0$ *and* $\mathtt{t}_1$*, the signal* $\mathtt{s}$ *is high*"... Thus we opted for representations 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.

4.1. Specification and Verification Strategy

Mealy and Moore automata [Mea55, Moo56] have been widely used for modeling circuit structures. It appeared to us that this notion is also an adequate and elegant model for representing behaviors that are described by the designers in an informal way. 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.

Automata are encoded as *recursive functions processing on co-inductive streams.* Let us mention that our encoding is quite general since it also applies to machines with an infinite number of states. Then, we introduce a notion of *equivalence* on these automata, so that establishing a circuit correctness amounts to proving that its structural automaton is equivalent to its behavioral automaton. The equivalence relation being co-inductive, the proof is performed co-inductively. It relies on a single generic lemma which is in fact a *proof schema* that underlies any correctness proof, and captures the temporal aspects of the reasoning.

The set of automata can be equipped with composition rules [HU79, Boo67]. We axiomatize this *algebraic structure* in CC and we use it to give modular descriptions of architectures. Thus, complex device verification processes are decomposed into the verification of simpler sub-components. We establish that the equivalence relation on automata is 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.

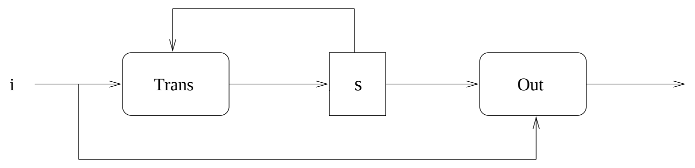

Fig. 5. Representation of a Mealy automaton

As it will be illustrated in section 5, 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 high level representations, that are depicted by clear transition diagrams, appears to be *more trustworthy* than other ones that are based on low level tools (like timing diagrams or low level automata). Moreover, the automata we handle have a *small number of states*. This is due to the fact that a lot of information is carried in the rich state data structures and by the transition and output functions (see section 5.3.2).

We also 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.

4.2. 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 (Fig. 5) is defined by 5 parameters as follows.

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`.

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)`

It is recursively defined as follows:

fix Mealy. λinp λs (Cons (Out inp_0 s) (Mealy inp' (Trans 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 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 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, as a fixpoint `States` of type `(Stream I)`→`S`→`(Stream S)`, defined as follows:

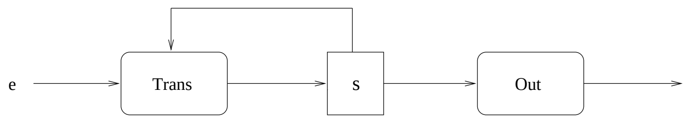

Fig. 6. Representation of a Moore automaton

`fix States.` λ`inp` λ`s (Cons s (States inp' (Trans inp`$_0$ `s))).`

One can notice that, with this encoding, the output stream `(Mealy inp s)` can be defined without referring to the state stream, contrary to what happens when infinite sequences are defined as functions on the natural numbers. Indeed, in this case, the output sequence $(\mathtt{o}_n)_{n:nat}$, and the state sequence $(\mathtt{s}_n)_{n:nat}$, are defined by :

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

Moore automata are described in Fig. 6. The Moore function associated with such an automaton is of type `(Stream I)`$\rightarrow$`S`$\rightarrow$`(Stream O)` and is defined similarly to the Mealy function. 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.

4.3. 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.

4.3.1. Parallel Composition

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

$$(\mathtt{I}_1,\ \mathtt{O}_1,\ \mathtt{S}_1,\ \mathtt{Trans}_1,\ \mathtt{Out}_1) \text{ and } (\mathtt{I}_2,\ \mathtt{O}_2,\ \mathtt{S}_2,\ \mathtt{Trans}_2,\ \mathtt{Out}_2)$$

and we define their Mealy functions

$$\mathtt{A}_1\ \mathtt{:=\ (Mealy\ Trans}_1\ \mathtt{Out}_1\mathtt{)} \text{ and } \mathtt{A}_2\ \mathtt{:=\ (Mealy\ Trans}_2\ \mathtt{Out}_2\mathtt{).}$$

In the following and when this does not introduce ambiguity, $\mathtt{A}_1$ and $\mathtt{A}_2$ will be employed to denote both the automata and their Mealy functions.

The parallel composition of $\mathtt{A}_1$ and $\mathtt{A}_2$ is described in Fig. 7. The two objects, on each side of the schema, need comments :
- $\mathtt{f=(f}_1\mathtt{,\ f}_2\mathtt{)}$ builds from the current input `i` the pair of inputs $\mathtt{(f}_1\mathtt{(i),\ f}_2\mathtt{(i))}$ for $\mathtt{A}_1$ and $\mathtt{A}_2$.
- `output` is of type $\mathtt{O}_1\times\mathtt{O}_2 \rightarrow\mathtt{O}$ and computes the global output from the outputs of $\mathtt{A}_1$ and $\mathtt{A}_2$.

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

```
parallel : (Stream I)→ S₁ → S₂ →(Stream O) :=
   λinp λs₁ λs₂ (Map output (Prod o₁ o₂)).
```

where, $\forall \mathtt{j}\in \{1,2\}$, $\mathtt{o_j\ =\ (A_j\ (Map\ f_j\ inp)\ s_j)}$ is the output stream of $\mathtt{A_j}$ for initial state $\mathtt{s_j}$. The global output stream is obtained by mapping the function `output` on the pointwise product of the output streams

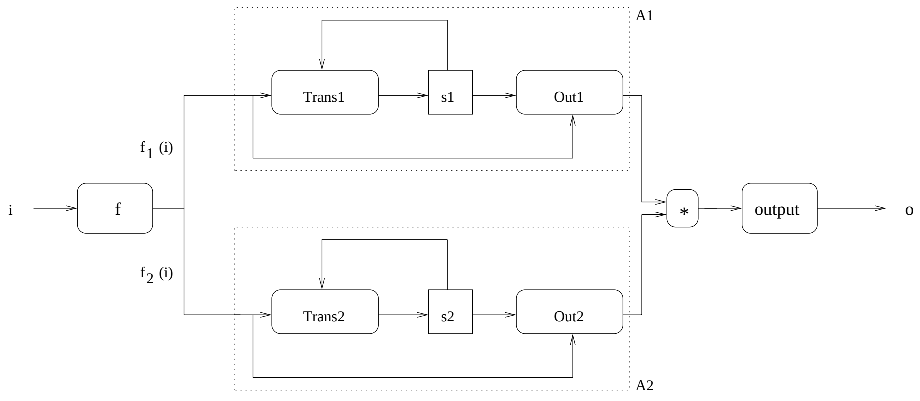

Fig. 7. Parallel Composition

of $\mathtt{A}_1$ and $\mathtt{A}_2$.

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 $\mathtt{A}_1$ and $\mathtt{A}_2$. Its transition function and its output function are respectively defined by:

$$
\begin{array}{l}
(\forall \mathtt{i:I})(\forall \mathtt{s}_1\mathtt{:S}_1)(\forall \mathtt{s}_2\mathtt{:S}_2)\\
\quad (\mathtt{Trans\_PC\ i\ (s_1,\ s_2))=((Trans_1\ (f_1\ i)\ s_1),\ (Trans_2\ (f_2\ i)\ s_2))}\\
\quad (\mathtt{Out\_PC\ i\ (s_1,\ s_2))=(output\ (Out_1\ (f_1\ i)\ s_1)\ (Out_2\ (f_2\ i)\ s_2)).}
\end{array}
$$

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

Lemma 2. Let `PC` be the Mealy function `(Mealy Trans_PC Out_PC)`. For all states $\mathtt{s}_1$ of type $\mathtt{S}_1$, $\mathtt{s}_2$ of type $\mathtt{S}_2$, and for all input stream `inp` of type `(Stream I)` the following relation holds:

$$\mathtt{(parallel\ inp\ s_1\ s_2)} \sim \mathtt{(PC\ inp\ (s_1,s_2))}$$

Proof. Let us clarify on this simple example how a proof term can be built by co-inductive reasoning. This proof consists in two steps:

(1) prove that the heads of the two streams are equal

(2) assuming that their tails are equivalent (co-induction hypothesis), deduce from (1) that the streams are equivalent.

For the reader familiar to type theory, we give the proof term. Its type being co-inductive, this term is a fixpoint in which the recursive call corresponds to applying the co-induction hypothesis. In the case of this lemma, it is particularly compact. Let us call it `Equiv_parallel_PC`. It is defined as follows:

$$\mathtt{fix\ Equiv\_parallel\_PC.}\ \lambda \mathtt{s}_1\ \lambda \mathtt{s}_2\ \lambda \mathtt{inp\ (eqS\ h\ (Equiv\_parallel\_PC\ s_1'\ s_2'\ inp')).}$$

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 $\mathtt{s}_1$' and $\mathtt{s}_2$' are the new states computed as follows:

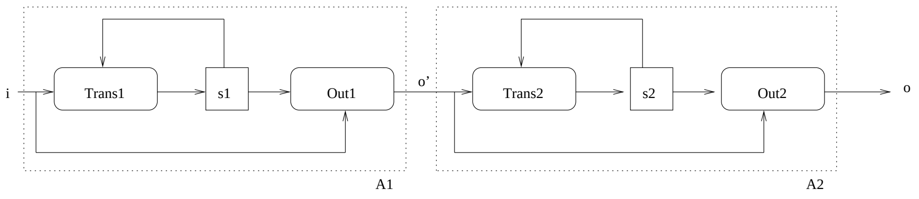

Fig. 8. Sequential Composition of two Mealy Automata

`s`$_1$`'= (Trans`$_1$` (f`$_1$` inp`$_0$`) s`$_1$`)` and `s`$_2$`'= (Trans`$_2$` (f`$_2$` inp`$_0$`) s`$_2$`)`

One can notice that Coq co-induction is easier to use than Park's co-induction principle, which requires to find an invariant. Moreover such co-inductive 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 on the natural numbers.

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 Trans`$_1$` Trans`$_2$` out`$_1$` out`$_2$` f output)` instead of `PC`. In the Coq proof assistant, 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. Outside a parameterized section, the parameters are discharged and appear explicitly in the terms that are defined in the section and that depend on the parameters.

4.3.2. Sequential Composition

Let us now consider two automata

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

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

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

The sequential composition is described in Fig. 8 and merely corresponds to the composition of the functions `A`$_1$ and `A`$_2$. 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 `A`$_1$ and a state of `A`$_2$. The transition and output functions of `SC` are defined by:

```
(∀i:I)(∀s₁:S₁)(∀s₂:S₂)
    (Trans_SC i (s₁, s₂))= ((Trans₁ i s₁), (Trans₂ (Out₁ i s₁) s₂))
    (Out_SC (s₁, s₂))= (Out₂ (Out₁ i s₁) s₂))
```

More precisely, we have the following lemma:

Lemma 3. 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.

4.3.3. 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. With-

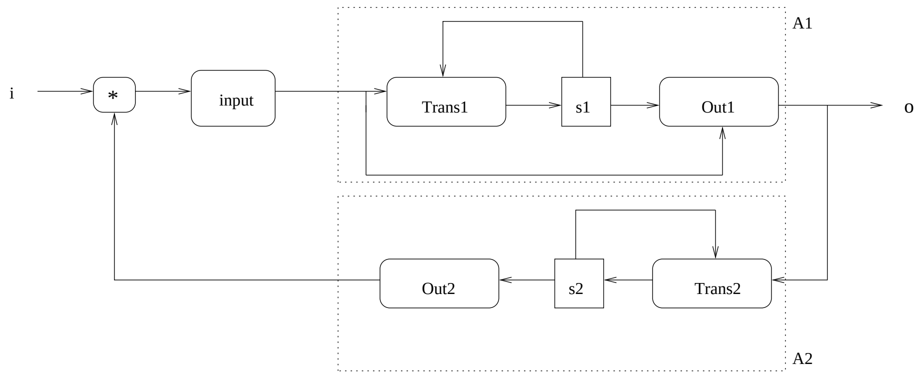

Fig. 9. Feedback Composition

out 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:

$$\texttt{(I}_1\texttt{, O, S}_1\texttt{, Trans}_1\texttt{, Out}_1\texttt{)} \text{ and } \texttt{(O, O}_2\texttt{, S}_2\texttt{, Trans}_2\texttt{, Out}_2\texttt{)}$$

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

$$\texttt{A}_1 \texttt{ := (Mealy Trans}_1\texttt{ Out}_1\texttt{)} \text{ and } \texttt{A}_2 \texttt{ := (Moore Trans}_2\texttt{ Out}_2\texttt{).}$$

Let `I` be the set of the feedback composition inputs. The function `input` of type $\texttt{I*O}_2 \rightarrow \texttt{I}_1$ (where the operator * denotes the cartesian product) is a parameter which computes the current input of $\texttt{A}_1$ from the current input `i` and the output of $\texttt{A}_2$. The following function `out` computes the current output of the device:

$\texttt{out:=}\ \lambda\texttt{i}\ \lambda\texttt{s}_1\ \lambda\texttt{s}_2\ \texttt{(Out}_1\texttt{ (input (i, (Out}_2\texttt{ s}_2\texttt{))) s}_1\texttt{)}$

The functions updating the states are defined by:

$\texttt{upd}_1\texttt{:=}\ \lambda\texttt{i}\ \lambda\texttt{s}_1\ \lambda\texttt{s}_2\ \texttt{(Trans}_1\texttt{ (input (i, (Out}_2\texttt{ s}_2\texttt{))) s}_1\texttt{)}$
$\texttt{upd}_2\texttt{:=}\ \lambda\texttt{i}\ \lambda\texttt{s}_1\ \lambda\texttt{s}_2\ \texttt{(Trans}_2\texttt{ (out i s}_1\texttt{ s}_2\texttt{) s}_2\texttt{)}$

We can now specify the `feedback` function that computes the output stream produced by the device represented in Fig. 9 with states having $\texttt{s}_1$ and $\texttt{s}_2$ as initial values, and in response to an input stream `inp`:

$\texttt{fix feedback.}\ \lambda\texttt{inp}\ \lambda\texttt{s}_1\ \lambda\texttt{s}_2\ \texttt{(Cons (out inp}_0\texttt{ s}_1\texttt{ s}_2\texttt{) (feedback inp' s}_1\texttt{' s}_2\texttt{'))}$

where $\forall\texttt{j}\in\{1,2\}$, $\texttt{s}_\texttt{j}\texttt{'= (upd}_\texttt{j}\texttt{ inp}_0\texttt{ s}_1\texttt{ s}_2\texttt{)}$ is the updated state of $\texttt{A}_\texttt{j}$.

As for the previous two rules, we show that this composition is equivalent to a Mealy automaton `FC` defined by the 5-uple $\texttt{(I,O, S}_1\texttt{*S}_2\texttt{, Trans\_FC, Out\_FC)}$ where:

$\texttt{(Trans\_FC i (s}_1\texttt{, s}_2\texttt{)) = ((upd}_1\texttt{ i s}_1\texttt{ s}_2\texttt{), (upd}_2\texttt{ i s}_1\texttt{ s}_2\texttt{)))}$
$\texttt{(Out\_FC i (s}_1\texttt{, s}_2\texttt{)) = (out i s}_1\texttt{ s}_2\texttt{)}$

The following lemma establishes the equivalence of the two devices.

Lemma 4. Let `FC` the Mealy function `(Mealy Trans_FC Out_FC)`. For all states $\texttt{s}_1$ of type $\texttt{S}_1$ and $\texttt{s}_2$ of type $\texttt{S}_2$, 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].

4.4. 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 on functions on streams.

Definition 5. 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: $A_1 \equiv A_2$ if and only if:

$(\forall s_1{:}S_1)(\exists s_2{:}S_2)(\forall inp{:}(Stream\ I))(A_1\ inp\ s_1) \sim (A_2\ inp\ s_2) \wedge$
$(\forall s_2{:}S_2)(\exists s_1{:}S_1)(\forall inp{:}(Stream\ I))(A_1\ inp\ s_1) \sim (A_2\ inp\ s_2).$

Lemma 6. The condition $A_1 \equiv A_2$ is equivalent to:

$(\forall s_1{:}S_1)(\exists s_2{:}S_2)(\forall inp_1,inp_2\ {:}(Stream\ I))\ (inp_1 \sim inp_2)\rightarrow(A_1\ inp_1\ s_1) \sim (A_2\ inp_2\ s_2) \wedge$
$(\forall s_2{:}S_2)(\exists s_1{:}S_1)(\forall inp_1,inp_2\ {:}(Stream\ I))\ (inp_1 \sim inp_2)\rightarrow(A_1\ inp_1\ s_1) \sim (A_2\ inp_2\ s_2).$

Proof. The proof consists in establishing first that for all automata A, for all states s, and for all equivalent input streams inp_1 and inp_2, $(A\ inp_1\ s) \sim (A\ inp_2\ s)$. Then one uses the transitivity of relation $\sim$.

Lemma 7. The relation $\equiv$ is an equivalence relation on the set of automata.

The proof is straightforward.

Lemma 8. The relation $\equiv$ over the set of automata is a congruence for the three interconnection rules.

That means that if A_1, B_1, A_2, and B_2 are automata, under the conditions over the input, output and state types for the interconnections to be correct, if $A_1 \equiv B_1$ and $A_2 \equiv B_2$ then $PC_{A1,A2} \equiv PC_{B1,B2}$, $SC_{A1,A2} \equiv SC_{B1,B2}$, and $FC_{A1,A2} \equiv FC_{B1,B2}$. Here $SC_{A1,A2}$, $PC_{A1,A2}$, and $FC_{A1,A2}$ denote the Mealy automata resulting from the sequential, parallel and feedback composition of A_1 and 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 9. Let A = (I, O, S, Trans, Out) be a Moore automaton and let B the Mealy automaton defined by B = (I, O, S, Trans, $(\lambda s{:}S)(\lambda i{:}I)(Out\ s)$). Then:

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

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

Lemma 10. 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' := $(\lambda i{:}I)(\lambda(s,o){:}S{*}O)((Trans\ i\ s),(Out\ i\ s))$ and
Out' := $(\lambda(s,o){:}S{*}O)o$.

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

$$(\forall i{:}I)(\forall inp{:}(Stream\ I))(\forall s{:}S)\ (A\ (Cons\ i\ inp)\ s) \sim (B\ inp\ (Trans\ i\ s)\ (Out\ i\ s))$$

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

4.5. 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`: $I \rightarrow S_1 \rightarrow S_2 \rightarrow Prop$ *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`: $S_1 \rightarrow S_2 \rightarrow Prop$ *and* `P`: $I \rightarrow S_1 \rightarrow S_2 \rightarrow Prop$ *be two predicates.* `R` *is invariant under* `P` *for the automata* A_1 *and* A_2, *if and only if:*

$(\forall i{:}I)(\forall s_1{:}S_1)(\forall s_2{:}S_2)\ (P\ i\ s_1\ s_2) \rightarrow (R\ s_1\ s_2) \rightarrow (R\ (Trans_1\ i\ s_1)\ (Trans_2\ i\ s_2)).$

Output relation. *Let* `R`: $S_1 \rightarrow S_2 \rightarrow Prop$ *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:*

$(\forall i{:}I)(\forall s_1{:}S_1)(\forall s_2{:}S_2)\ (R\ s_1\ s_2) \rightarrow (Out_1\ i\ s_1) = (Out_2\ i\ s_2).$

We can now set forth the equivalence lemma:

Lemma 11. Let `P`: $I \rightarrow S_1 \rightarrow S_2 \rightarrow Prop$ and `R`: $S_1 \rightarrow S_2 \rightarrow Prop$ be two predicates. Let us assume that `R` is an output relation that is invariant under `P`, and that holds on two initial states s_1 and s_2. Then, for all input streams `inp`:

$$(inv\ P\ inp\ SS_1\ SS_2) \rightarrow (A_1\ inp\ s_1) \sim (A_2\ inp\ s_2)$$

where $\forall j \in \{1,2\}$, $SS_j = (States\ Trans_j\ Out_j\ inp\ s_j)$ is the stream of the states of A_j.

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 Real Circuit

In this section, we outline the certification process of a realistic circuit. It illustrates our methodology and gives evidence of its feasibility.

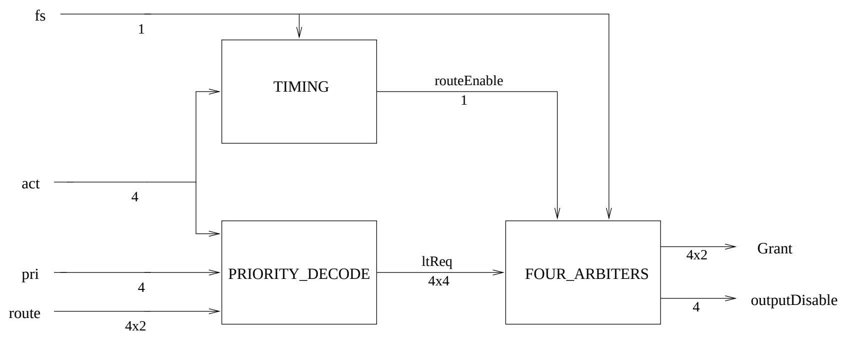

Fig. 10. Arbitration Unit

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 the central part of an experimental local area network based on Asynchronous Transfer Mode (ATM). This communication mode consists in sending data on a network by creating virtual connections. Each machine communicates with others via its switch. The main role of such a device is performing switching of data from four input ports to four 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 data to be sent, and then it performs arbitration. It is the interconnection of 3 modules (Fig.10) :

- `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 4.5, we present in detail the verification of `TIMING`, which is rather simple but 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.11.

5.2.1. Structural Description

The structure of `TIMING` corresponds exactly to a Mealy automaton. Its transition function represents the boxed combinatorial part in Fig. 11. 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
S := (d_list bool 2).
```

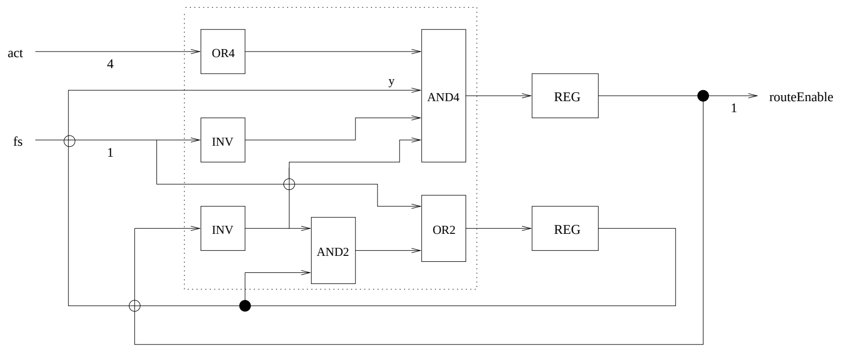

Fig. 11. Timing Unit

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

```
INV := neg.
AND₄ := (λa,b,c,d:bool) (andb a (andb b (andb c d))).
```

The automaton representing the circuit in Fig. 11 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(s₁, s₂) (List2 (AND₄ (OR₄ act) s₂ (INV fs) (INV s₁))
                                                    (OR₂ fs (AND₂ (INV s₁) s₂)))

Out_Timing:I -> S -> O := λi λ(s₁, s₂) s₁

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. Behavioral Description

The expected behavior is presented in Fig. 12. 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 defined by cases. The automaton `Behavior_TIMING` is obtained as usual by an instantiation of `Mealy`.

5.2.3. Proof of Correctness

The proof of correctness relies on the notions introduced in section 4.5. To prove the equivalence between `Behavior_TIMING` and `Structure_TIMING` we apply lemma 11. For that, we have to define a relation between the state `(s₁, s₂)` of the structure and the state `s` of the behavior. This relation, namely `R_Timing`, expresses that the first register value `s₁` 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 `s₂` of the structure.

```
R_Timing:S_beh→S→Prop := λs λ(s₁, s₂)
                          ((∀i:I)s₁=(Out_T i s)) ∧ (s = START_t ∧ s₂ = false ∨
                                                    s = WAIT_t ∧ s₂ = true ∨
```

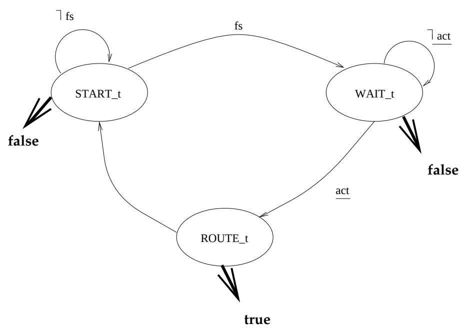

Fig. 12. Timing Behavior

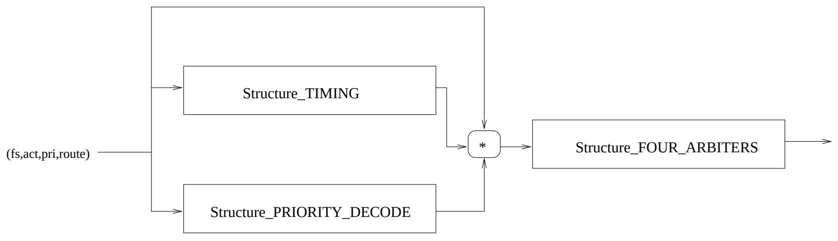

Fig. 13. The Arbitration Structure as an Interconnection of Automata

s = ROUTE_t ∧ s_2 = 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 12. `(∀inp:(Stream I))(∀s:S)(∀s_beh:S_beh)`
`(R_Timing s_beh s)→ (Behavior_TIMING inp s_beh)∼(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 lemma 11.

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. 10). Its structure is described in Fig. 13.

5.3.1. Structural Description

The structural description is given in several steps (Fig. 14, 15, 16), by using the parallel and sequential composition rules (`PC` and `SC`) on automata presented in section 4.3.

For example, the final definition representing the structure of `ARBITRATION` as a sequential composition of two intermediate automata (Fig. 16) is specified by the following definitions. The input type is:

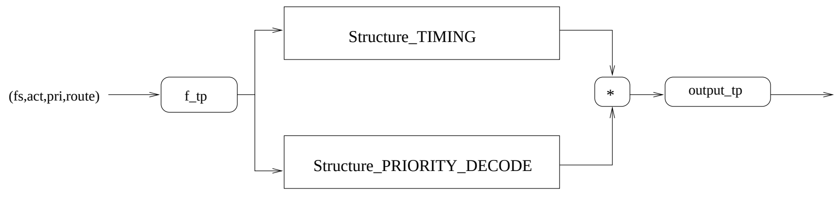

Fig. 14. *Structure_TIMINGPDECODE*

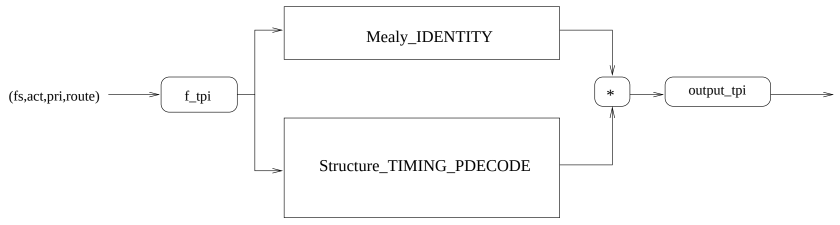

Fig. 15. *Structure_TIMINGPDECODE_ID*

```
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)`$\longrightarrow$`S`$\longrightarrow$`(Stream O) := (SC Trans`$_{\mathtt{TPD}}$ `Trans`$_{\mathtt{FA}}$ `Out`$_{\mathtt{TPD}}$ `Out`$_{\mathtt{FA}}$`).`

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

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. 17. 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),

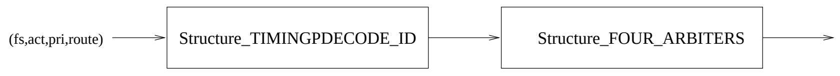

Fig. 16. *Structure_ARBITRATION*

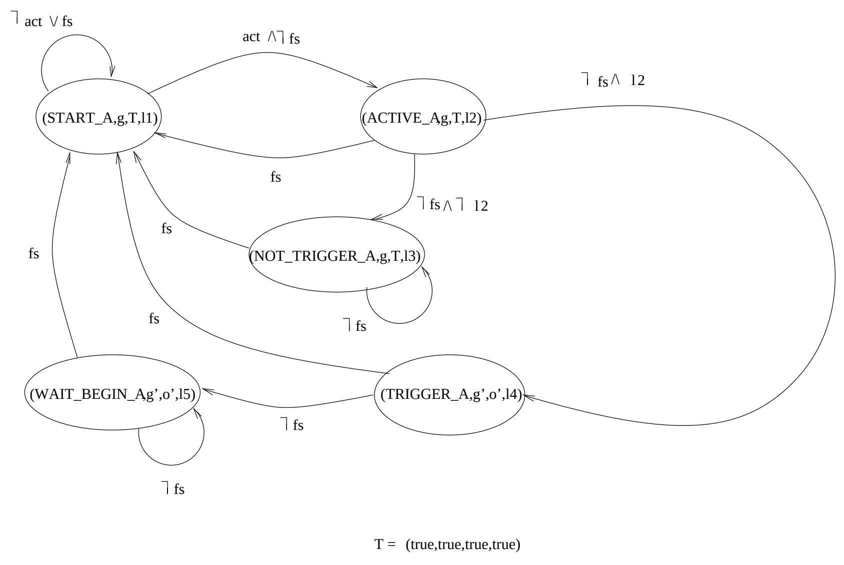

Fig. 17. Arbitration Behavior

- 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 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_ARBITRATION`. 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 lemma 11 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. Discussion and Related Work

As far as related work is concerned, the study closest to ours is that of Curzon, who performs the verification of the ATM Switch Fabric with the HOL theorem prover [Cur94a, Cur94b]. It was an helpful starting point for our investigations despite the fact that his approach is completely different. Let us make a comparison on the example of the Timing unit.

Here is the behavioral representation given by Curzon as a temporal statement. We show it here just to give an idea of the style of description, not to be understood in detail.

```
∀ fs, act, re. TIMING_BEH((fs, act), re)=
(∀ ts ta te.∼(fs 0)∧(∀t. ∼(fs (t+1))∨ ∼(EXISTABIT I (act t)))→
      (∀ t. ∼(fs t ∧ re t)) ∧
      (AFRAME2 ts ta te fs act → STABLE (ts+1) (ta+1) re F ∧
                                 (re (ta+1) = T) ∧
                                 (STABLE (ta+2) (te+1) re F) ∧
                                 (IFRAME ts te fs act →(STABLE (ts+1) (te+1) re F))
```

In this statement, `ts`, `ta`, `te`, `t` are natural numbers that represent time points. The signals `fs`, `re`, `act` are functions on the natural numbers. Such a description is to be compared to the small automaton depicted in Fig.12.

To establish the correctness of this unit, Curzon proves an implication between the structural specification and this statement (whereas we prove an equivalence). For that, he introduces a more concrete description `TIMING_BEH2`, closer to the implementation. First, it is proved that the structure implements `TIMING_BEH2`, and then that `TIMING_BEH2` implies `TIMING_BEH`. This requires establishing preliminarily that `(∀ t. ∼(fs t ∧ re t))`. This is done by induction on parameter `t`. In fact, as in general the goals are universal statements on time points, several (sometimes nested) inductions are required. This is to be compared with our correctness lemma which is an immediate consequence of lemma 11. The effort is focused on establishing that a relation on the states is an *output relation* and that it is *invariant.* This is quite simple and purely combinatorial.

This small example illustrates the advantage of getting rid of time parameters and adopting an abstract algebraic way of specifying. The more challenging the device to be verified, the more important the benefit. The behavior of the Arbitration unit is quite complex. However we could model it as a 5 states automaton.

Moreover, HOL does not provide dependent types and Curzon reported that some errors in his specifications caused a big loss of time in his verification process. By working with Coq, we could give more precise encodings and thus avoid such problems. It is worth noticing that a significant part of our results, related to automata and dependent types libraries, are reusable and available for hardware verifiers. Let us emphasize that the axiomatization of automata provides a co-inductive notion of congruence. Since it also covers the case when the set of states is infinite, it applies to systems more general than circuits.

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 performed classically 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 is still) 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. 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]. More recently, Kort and al. proposed an hybrid method for using both HOL and MDG [KSC03]. The user must give a modular description of the device and must provide a behavioral description of its sub-modules. If the circuit cannot be proved automatically by MDG, due to state explosion, the correctness theorem is split by HOL into subgoals related to the correctness of its sub-modules. Then, one tries again to have these subgoals automatically established by MDG. Although it offers more automation, this approach has some drawbacks. First, HOL must trust MDG. Second, it requires more work in preparing the data, since MDG handles lower level encodings. It also requires the user to provide behaviors for the sub-modules in case MDG fails in verifying a too big device. Third, the MDG proof process may not terminate. In this case, some heuristics must be applied, by hand.

7. Conclusion

In this paper, we have thoroughly investigated how to take advantage of the expressiveness of the Calculus of (Co)-Inductive Constructions (CC) in the field of hardware verification. As far as modeling circuits is concerned, our approach relies on the use of dependent types, to accurately describe combinatorial features, and on the use of co-inductive types to take into account the temporal aspects of synchronous sequential circuits.

As handling dependent types requires expertise, we have provided several general libraries that may be of interest for verifiers. Moreover, we have demonstrated that this way of specifying can be used significantly in practice. We have shown that technical problems can be overcome both on a small example that involves various kinds of dependent types, and on a real circuit such the *ATM switch fabric*. We also have presented, as a first step in circuit synthesis, a short experiment on the Coq extraction mechanism.

Then, we have proposed a high-level generic methodology, for verifying synchronous sequential circuits in Coq. It is based on a uniform co-inductive description of the structures and the expected behaviors of circuits by means of Mealy automata. This algebraic approach makes it possible to get rid of time parameters and leads to clearer and more elegant descriptions. Moreover, establishing the correctness of sequential circuits with theorem provers, is generally done by proving a collection of lemmas by (sometimes nested) induction on the steps of time. Thus, the temporal considerations are omnipresent all along the proof. With our approach, we capture once and for all in one generic lemma 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.

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 compact behavioral automata. 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.

As illustrated in section 5.3, and due to the congruence relation we have defined on automata, the correctness of a unit is obtained from that of its components. 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.

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

It would now be interesting to investigate how to make the combinatorial parts of the proof more automatic. Indeed, this is the tedious aspect of our work. Working with Coq requires expertise and is rather time consuming. The general methodology we have proposed is accompanied by reusable libraries. Jointly with a way for automatizing combinatorial proofs, we think that it would be a very convenient tool for hardware verification

The Coq files related to this paper can be found in [CJa] and [CJb].

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.

[CJa] Solange Coupet-Grimal and Line Jakubiec. Verification and synthesis of hardware linear arithmetic structures. *The Coq Users's Contributions.* http://coq.inria.fr/contribs-eng.htlm.

[CJb] Solange Coupet-Grimal and Line Jakubiec. Verification of sequential synchronous circuits. http://www.lif.univ-mrs.fr/∼solange/CONTRIB_HARDWARE/FAIRISLE.

[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.

[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.

[HD92] F.K. Hanna and N. Daeche. Dependent types and formal synthesis. *Phil. Trans. R. Soc. Lond.*, 339:121–135, 1992.

[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.

[KSC03] Sofiène Kort Skander, Tahar and Paul Curzon. Hierarchical formal verification using a hybrid tool. *International Journal Softw Tools Technol Transfer*, 4:313–322, 2003.

[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.