

Testing of Abstract Components

Bilal Kanso, Marc Aiguier, Frédéric Boulanger, Assia Touil

▶ **To cite this version:**

Bilal Kanso, Marc Aiguier, Frédéric Boulanger, Assia Touil. Testing of Abstract Components. ICTAC 2010 - International Conference on Theoretical Aspect of Computing., Sep 2010, Brazil. pp.184-198. hal-00782893

HAL Id: hal-00782893

https://centralesupelec.hal.science/hal-00782893

Submitted on 11 Apr 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Testing of Abstract Components

Bilal Kanso[1,2], Marc Aiguier[1], Fréderic Boulanger[2], Assia Touil[2]

[1] École Centrale Paris
Laboratoire de Mathématiques Appliquées aux Systèmes (MAS)
Grande Voie des Vignes F-92295 Châtenay-Malabry
email : {marc.aiguier, bilal.kanso}@ecp.fr

[2] SUPELEC Systems Sciences (E3S) - Computer Science Department
3 rue Joliot-Curie F-91192 Gif-sur-Yvette cedex
email : {frederic.boulanger, assia.touil}@supelec.fr

Abstract. In this paper, we present a conformance testing theory for Barbosa's abstract components. This is made possible by defining first a trace model for components by causal transfer functions, that is functions of dataflow transformations rythmitical by discrete instants. This then allows us to exploit a particular analysis technique to define a test selection strategy based on test purposes defined as some subtrees of the execution tree built from component traces. Moreover, we show in this paper that Barbosa's definition of components is abstract enough to subsume a large family of state-base formalisms such as Mealy automata, Labeled Transition Systems (LTS), Input Output Labeled Transition Systems (IOLTS), etc. by instantiating the monads underlying Barbosa's definition. Hence, the conformance theory presented in this paper is *de facto* a generalization of standard ones we find for different state-base formalisms.
Keywords: Component based system, Coalgebra, Monad, Trace semantics, Transfert function, Conformance testing, Test purpose.

Introduction

Complex industrial systems are built with several components that are designed into different models. These components are integrated altogether to interact in order to a coherent way. To deal with these heterogeneous components from a global point of view, we model them with coalgebras. In this framework, an important work has been done by Barbosa, who has introduced state-based software components as concrete coalgebras for some set endofunctors [1, 19]. The interest of such modeling is twofold: First, Barbosa has defined a component as any coalgebra over the endofunctor $\mathcal{H} = T(Out \times _)^{In}$ where T is a monad[3], In and Out are two sets elements of which denote respectively component input and output. Hence, Barbosa's definition of component is an extension of Mealy automata [7, 18] that have been shown efficient to specify component behaviors

[3] All the definitions and notations of coalgebras and monads are recalled in Section 1 of this paper.

deterministically. Here, using monads allows ones to abstract away from determinism. Indeed, monads have been introduced in [22] to generically consider a wide range of computation structures such as partiality, non-determinism, etc. Then, Barbosa's definition of component allows to define component independently of any computation structure. In this paper, we will go further than this definition that unifies in a same framework a large family of state-base formalisms such as Mealy automata, Labeled Transition Systems [5, 20], Input-Iutput Symbolic Sransitions [9, 10, 14], etc. Second, following Rutten's works [11, 26], defining component behaviors as an extension of Mealy automata to any computation structure by using monads, will allow us to define a trace model over components by causal transfer functions, that is functions of dataflow transformations of the form: $y = \mathcal{F}(x, q, t)$ where x, y and q are respectively, the input, output and state of the component under consideration, and t stands for time considered here as discrete.

This last point warms up our contribution in this paper. Indeed, defining a trace model from causal functions will allow us: Firstly, to give some results about the important notion in the categorical theory of coalgebras of final coalgebras. Hence, we will show the existence of such a final coalgebra in the category of coalgebras over a signature $T(Out \times _)^{In}$ under some sufficient conditions on the monad T. The interest of such results is there a powerful reasoning principle that underlies final coalgebras, which is coinduction. Secondly, to define a conformance testing theory for components, that is the most important contribution of this paper. Hence, following some previous works that have been done by some authors of this paper [10], test purposes will be defined as some particular subtrees of the execution tree built from our trace model for components. We will then define an algorithm which from test purposes will generate test cases. As in [10], this algorithm will be given by a set of inference rules. Each rule is dedicated to handle an observation from the system under test (SUT) or a simulation sent by the test case to the SUT. This testing process leads to a verdict.

The paper is then structured as follows: In Section 1, we recall the basic notions of the categorical theory of coalgebras and monads that will be useful in this paper. Then, in Section 2, we recall Barbosa's definition of components and we define over, a trace model from causal transfer functions. We will take the benefit to have formalized components as coalgebras to extend some standard results connected to the definition of a terminal component. In Section 3, we present our conformance testing theory for components. Finally, Section 4 presents on-the-fly rules for generating test cases.

1 Preliminaries

This paper relies on many terms and notations from the categorical theory of coalgebras and monads. In this section, we briefly introduce these notions and notations used in the rest of the paper. Interested readers can refer textbooks such as [2, 8, 17].

1.1 Categories, functors and natural transformations

A **category** $\mathbb{C}$ is a mathematical structure consisting of a collection of objects $\mathsf{Obj}(\mathbb{C})$ and a collection of maps or morphisms $\mathsf{Hom}(\mathbb{C})$. Each map $f : X \to Y$ has a domain $X \in \mathsf{Obj}(\mathbb{C})$ and a codomain $Y \in \mathsf{Obj}(\mathbb{C})$.

Maps may be composed using the $\circ$ operation, which is associative. For each object $X \in \mathsf{Obj}(\mathbb{C})$, there is an identity map $\mathsf{id}_X : X \to X$ which is neutral for the $\circ$ operation: for any map $f : X \to Y$, one has $f \circ \mathsf{id}_X = f = \mathsf{id}_Y \circ f$.

An object $I \in \mathsf{Obj}(\mathbb{C})$ is initial if for any object $X \in \mathsf{Obj}(\mathbb{C})$, there is a unique morphism $f : I \to X$ in $\mathsf{Hom}(\mathbb{C})$. Conversely, an object $F \in \mathsf{Obj}(\mathbb{C})$ is final if for any object $X \in \mathsf{Obj}(\mathbb{C})$, there is a unique morphism $f : X \to F$ in $\mathsf{Hom}(\mathbb{C})$.

Given two categories $\mathbb{C}$ and $\mathbb{D}$, a **functor** $F : \mathbb{C} \to \mathbb{D}$ consists of two mappings $\mathsf{Obj}(\mathbb{C}) \to \mathsf{Obj}(\mathbb{D})$ and $\mathsf{Hom}(\mathbb{C}) \to \mathsf{Hom}(\mathbb{D})$, both written F, such that:

- F preserves domains and codomains:
  if $f : X \to Y$ is in $\mathbb{C}$, $F(f) : F(X) \to F(Y)$ is in $\mathbb{D}$
- F preserves identities: $\forall X \in \mathbb{C}, F(\mathsf{id}_X) = \mathsf{id}_{F(X)}$
- F preserves composition:
  $\forall f : X \to Y$ and $g : Y \to Z$ in $\mathbb{C}$, $F(g \circ f) = F(g) \circ F(f)$ in $\mathbb{D}$.

Given two functors $F, G : \mathbb{C} \to \mathbb{D}$ from a category $\mathbb{C}$ to a category $\mathbb{D}$, a **natural transformation** $\varepsilon : F \Rightarrow G$ associates to any object $X \in \mathbb{C}$ a morphism $\varepsilon_X : F(X) \to G(X)$ in $\mathbb{D}$, called the component of ε at X, such that for every morphism $f : X \to Y$ in $\mathbb{C}$, we have $\varepsilon_Y \circ F(f) = G(f) \circ \varepsilon_X$.

1.2 Algebras and coalgebras

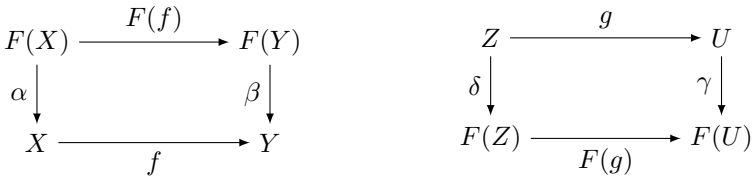
Given an endofunctor $F : \mathbb{C} \to \mathbb{C}$ on a category $\mathbb{C}$, an **F-algebra** is defined by a carrier object $X \in \mathbb{C}$ and a morphism $\alpha : F(X) \to X$. In this categorical definition, F gives the signature of the algebra. For instance, with $\mathbf{1}$ denoting the singleton set $\{\star\}$, if we consider the functor $F = \mathbf{1} + _$ which maps $X \mapsto \mathbf{1} + X$, the F-algebra $(\mathbb{N}, [0, \mathsf{succ}])$ is Peano's algebra of natural numbers, with the usual constant $0 : \mathbf{1} \to \mathbb{N}$ and constructor $\mathsf{succ} : \mathbb{N} \to \mathbb{N}$.

Similarly, an **F-coalgebra** is defined by a carrier object $X \in \mathbb{C}$ and a morphism $\alpha : X \to F(X)$. In the common case where $\mathbb{C}$ is **Set**, the category of sets, the signature functor of an algebra describes operations for building elements of the carrier object. On the contrary, in a coalgebra, the signature functor describes operations for observing elements of the carrier objet. For instance, a Mealy machine can be described as a F-coalgebra $(S, \langle \mathsf{out}, \mathsf{next} \rangle)$ of the functor $F = (Out \times _)^{In}$ with S, In and Out as its sets of states, input and output respectively.

1.3 Induction and coinduction

An homomorphism of (co)algebras is a morphism from the carrier object of a (co)algebra to the carrier object of another (co)algebra which preserves the

structure of the (co)algebras. On the following commutative diagrams, f is an homomorphism of algebras and g is an homomorphism of coalgebras:



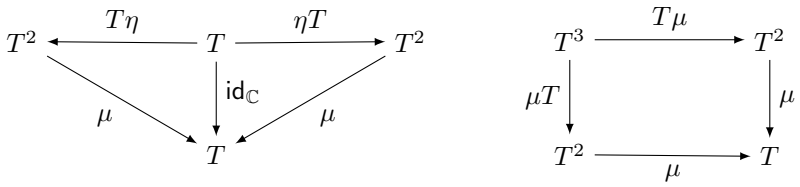
F-algebras and homomorphisms of algebras constitute a category $\mathbf{Alg}(F)$. Similarly, F-coalgebras and homomorphisms of coalgebras constitute a category $\mathbf{CoAlg}(F)$. If an initial algebra exists in $\mathbf{Alg}(F)$, it is unique, and its structure map is an isomorphism. The uniqueness of the homomorphism from an initial object to the other objects of a category is the key for defining morphisms by induction: giving the structure of an F-algebra (X, β) defines uniquely the homomorphism $f : I \to X$ from the initial F-algebra (I, α) to this algebra.

Conversely, if a final coalgebra exists in $\mathbf{CoAlg}(F)$, it is unique, and its structure map is an isomorphism. The uniqueness of the homomorphism from any object to a final object of a category is the key for defining morphisms by coinduction: giving the structure of an F-coalgebra (Y, δ) defines uniquely the morphism $f : Y \to F$ from this coalgebra to the final F-coalgebra (F, ω).

An interesting property is that if F is a finite Kripke polynomial functor, $\mathbf{Alg}(F)$ has an initial algebra and $\mathbf{CoAlg}(F)$ has a final coalgebra. Finite Kripke polynomial functors are endofunctors of the category **Set** which include the identity functor, the constant functors, and are closed by product, coproduct, exponent (or function space), and finite powerset.

1.4 Monads

Monads [17] are a powerful abstraction for adding structure to objects. Given a category $\mathbb{C}$, a **monad** consists of an endofunctor $T : \mathbb{C} \to \mathbb{C}$ equipped with two natural transformations $\eta : \mathsf{id}_{\mathbb{C}} \Rightarrow T$ and $\mu : T^2 \Rightarrow T$ which satisfy the conditions $\mu \circ T\eta = \mu \circ \eta T = \mathsf{id}_{\mathbb{C}}$ and $\mu \circ T\mu = \mu \circ \mu T$:



η is called the *unit* of the monad. Its components map objects in $\mathbb{C}$ to their naturally structured counterpart. μ is the *product* of the monad. Its components map objects with two levels of structure to objects with only one level of structure. The first condition states that a doubly structured object $\eta_{T(X)}(t)$ built by η from a structured object t is flattened by μ to the same structured object as a structured object $T(\eta_X)(x)$ made of structured objects built by η. The second

condition states that when flattening two levels of structure, we get the same result by flattening the outer structure first (with $\mu_{T(X)}$) or the inner structure first (with $T(\mu_X)$).

Let us consider a monad built on the powerset functor $\mathcal{P} : \mathbf{Set} \to \mathbf{Set}$. It can be used to model non-deterministic state machines by replacing the target state of a transition by a set of possible target states. The component $\eta_S : S \to \mathcal{P}(S)$ of the unit of this monad at state space S has to build a set of states from a state. We can then choose $\eta_S : \sigma \mapsto \{\sigma\}$. The component $\mu_S : \mathcal{P}(\mathcal{P}(S)) \to \mathcal{P}(S)$ of the product of the monad at state space S has to flatten a set of sets of states into a set of states. For a series of sets of states (s_i), $\forall i, s_i \in \mathcal{P}(S)$, we can then choose $\mu_S : \{s_1 \dots s_i \dots\} \mapsto \cup s_i$.

Moreover, monads have also been used to represent many computation situations such as partiality, side-effects, exceptions, *etc* [22] . For instance, partiality can be represented by the monad $T : S \to S \cup \{\bot\}$ equipped with both obvious natural transformations η and μ which for any set S are defined by:

$$\eta_S : s \mapsto s \quad \text{and} \quad \mu_S : \begin{cases} \bot \mapsto \bot \\ s \mapsto s \end{cases}$$

2 Transfer functions and components

In this section, we will use the definition given by Barbosa in [1, 19] to define components, *i.e* as coalgebras of the **Set** endofunctor $T(Out \times _)^{In}$ where In and Out are the sets of respectively input and output data and T is a monad. As we will see in Section 2.2, the interest of Barbosa's definition of component is it large enough to consider generically the notion of component, but also to unify in a same framework a large family of formalisms classically used to specify state-based systems such as *Mealy machines* [7, 18], *Labelled Transition Systems (LTS)* [5, 20], *Input-Output Labelled Transition Systems (IOLTS)* [6, 29], *etc.*

Similarly to the Rutten's works in [11, 26], we will denote component's behaviors by a transfer function.

2.1 Transfer function

In the following, we will note ω the least infinite ordinal, identified with the corresponding hereditarily transitive set.

Definition 1 (Dataflow). *A **dataflow** over a set of values A is a mapping $x : \omega \to A$. The set of all dataflows over A is noted A^ω.*

As we will see in the next section, the observable behavior of components will be described by its associated transfer function. Transfer functions can be seen as dataflow transformers satisfying the causality condition in a standard framework [27], that is the output data at index n only depends on input data at indexes $0, \dots, n$.

Definition 2 (Transfer function). *Let T be a monad. Let In and Out be two sets denoting, respectively, the input and output domains. A function $\mathcal{F} : In^\omega \longrightarrow Out^\omega$ is a* ***transfer function*** *if, and only if it is causal, that is:*

$$\forall n \in \omega, \forall x, y \in In^\omega, (\forall m, 0 \leq m \leq n, x(m) = y(m)) \Longrightarrow \mathcal{F}(x)(n) = \mathcal{F}(y)(n)$$

2.2 Components

Definition 3 (Components). *Let In and Out be two sets denoting, respectively, the values in input and in output. Let T be a monad. A* ***component*** *$\mathcal{C}$ is any coalgebra (S, α) for the signature $\mathcal{H} = T(Out \times _)^{In} : \mathbf{Set} \to \mathbf{Set}$ with a distinguished element s_0 denoting the initial state of the component $\mathcal{C}$.*

Definition 4 (Category of components). *Let $\mathcal{C}$ and $\mathcal{C}'$ be two components over $\mathcal{H} = T(Out \times _)^{In}$. A* ***component morphism*** *$h : \mathcal{C} \to \mathcal{C}'$ is a coalgebra homomorphism $h : (S, \alpha) \to (S', \alpha')$ such that $h(s_0) = h(s'_0)$.*
We note $\mathbf{Cat}(\mathcal{H})$ *the* ***category of systems*** *over $\mathcal{H}$.*

The definition of components in Definition 3 allows to unify in a same framework a large family of formalisms classically used to specify state-based systems such as *Mealy machines*, *LTS* and *IOLTS*. Hence, making T the identity functor $\mathcal{I}d$, the resulting component corresponds to *Mealy machines*. Taking $In = \{\}$ and $Out = Act$ as a set of symbols standing for actions names, and instantiating T with the powerset functor $\mathcal{P}$, the resulting component leads to labelled transition systems. Finally, taking the monad T as the powerset monad $\mathcal{P}$ and by imposing the supplementary property on the transition function $\alpha : S \longrightarrow \mathcal{P}(Out \times S)^{In}$:

$$\forall i \in In, \forall s \in S, (o, s') \in \alpha(s)(i) \Longrightarrow \text{ either } i = \epsilon \text{ or } o = \epsilon$$

leads to $IOLTS$.

Example 1. We illustrate the notions and results previously mentioned with the simple example of a coffee machine $\mathcal{M}$ presented as the transition diagram shown on Figure 1. The behavior of $\mathcal{M}$ is the following: from its initial state STDBY, when it receives a coin from the user, the machine goes into the READY state. Then, when the user presses the "coffee" button, the machine either serves a coffee to the user and goes to the STDBY state, or it fails to do so, refunds the user and goes to the FAILED state. The only escape from the FAILED state is to have a repair. In our framework, this machine is considered as a component $\mathcal{M} = (S, s^0, \alpha)$ over the signature $\mathcal{P}_f(Out \times _)^{In}$. The state space[4] is $S = \{\text{STDBY}, \text{READY}, \text{FAILED}\}$ and $s_0 = \text{STDBY}$. The set of inputs is $In = \{\text{coin}, \text{coffee}, \text{repair}\}$ and the set of outputs is $Out = \{\bot, \text{served}, \text{refund}\}$. Finally, the transition function: $\alpha : S \longrightarrow \mathcal{P}_f\big(\{\bot, \text{served}, \text{refund}\} \times S\big)^{\{\text{coin},\text{coffee},\text{repair}\}}$ is defined as follows :

$$\begin{cases} \alpha(\text{STDBY})(\text{coin}) = \big\{(\bot, \text{READY})\big\} \\ \alpha(\text{READY})(\text{coffee}) = \big\{(\text{served}, \text{STDBY}), (\text{refund}, \text{FAILED})\big\} \\ \alpha(\text{FAILED})(\text{repair}) = \big\{(\bot, \text{STDBY})\big\} \end{cases}$$

[4] $\mathcal{P}_f(X) = \{U \subseteq X | U \text{ is finite}\}$ is the finite powerset of X.

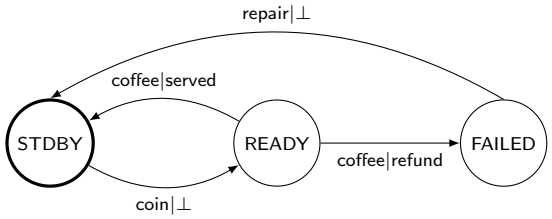


Fig. 1. Coffee machine

2.3 Traces

To associate behaviors to components by their transfer function, we need to impose the supplementary condition on the monad T that there exists a natural transformation $\eta^{-1} : T \Longrightarrow \mathcal{P}$ where $\mathcal{P} : S \mapsto \mathcal{P}(S)$ is the powerset functor, such that $\forall S \in \mathbf{Set}, \forall s \in S, \eta_S^{-1}(\eta_S(s)) = \{s\}$.

Most monads used to represent computation situations satisfy the above condition. For instance, for the monad $T : S \mapsto \mathcal{P}(S)$, η_S^{-1} is the identity on sets, while for the functor $T : S \mapsto S \cup \{\bot\}$, η_S^{-1} is the mapping that behaves like η_S for every element $s \in S$ and associates the emptyset for $\bot$. The interest of η^{-1} is to allow the association of a set of transfer functions to a component (S, α) as its possible traces. Indeed, we need to "compute" for a sequence $x \in In^\omega$ all the outputs o after "performing" any sequence of states $(s_0, \ldots, s_k)$ such that s_j is obtained from s_{j-1} by $x(j-1)$. However, we do not know how to characterize s_j with respect to $\alpha(s_{j-1})(x(j-1))$. The problem is that nothing ensures that elements in $\alpha(s_{j-1})(x(j-1))$ are couples (output, state). Indeed, the monad T takes the product of a set of output Out and a set of states S and yields another set which may not have the structure of $Out \times S$. The mapping $\eta^{-1}_{Out\times S}$ maps back to this structure. Elements in $\eta^{-1}_{Out\times S}(\alpha(s_{j-1})(x(j-1)))$ are couples (output, state). In the following, we note $\eta^{-1}_{Out\times S}(\alpha(s)(i))_{|_1}$ (*resp.* $\eta^{-1}_{Out\times S}(\alpha(s)(i))_{|_2}$) the set composed of all first arguments (*resp.* second arguments) of couples in $\alpha(s)(i)$. Hence, component traces are defined as follows:

Definition 5 (Component traces).
Let $\mathcal{C}$ be a component over $\mathcal{H} = T(Out \times _)^{In}$. The **Traces** *from a state s of $\mathcal{C}$ is the whole set of transfer functions $\mathcal{F}_s : In^\omega \to Out^\omega$ defined for every $x \in In^\omega$ such that there exists an infinite sequence of states $s_0, s_1, \ldots, s_k, \ldots \in S$ with $s_0 = s$ and satisfying: $\forall j \geq 1, s_j \in \eta^{-1}_{Out\times S}(\alpha(s_{j-1})(x(j-1)))_{|_2}$ and for every $k \in \omega$, $\mathcal{F}_s(x)(k) = o_k$ such that $(o_k, s_{k+1}) \in \eta^{-1}_{Out\times S}(\alpha(s_k)(x(k)))$*
*Hence, $\mathcal{C}$***'s traces** *are the set of transfer functions $\mathcal{F}_{s_0}$ as defined above.*

In the context of our work, we are mainly interested by finite traces. Finite traces are then any finite sequence of couples (input|output) defined as follows :

Definition 6 (Component finite traces). *Let $\mathcal{F}_{s_0}$ be a trace of a component $\mathcal{C}$, let $n \in \mathbb{N}$. The* ***finite trace*** *of length n $\mathcal{F}_{s_{0|_n}}$ associated to $\mathcal{F}_{s_0}$ is the whole set of the finite sequence $\langle i_0|o_0, \ldots, i_n|o_n\rangle$ such that there exists $x \in In^\omega$ where for every j, $0 \leq j \leq n$, $x(j) = i_j$, and $\mathcal{F}_{s_0}(x(j)) = o_j$.*
Then, $Trace(\mathcal{C}) = \bigcup_{\mathcal{F}_{s_0}} \bigcup_{n\in\mathbb{N}} \mathcal{F}_{s_{0|_n}}$ defines the whole set of finite traces over $\mathcal{C}$.

2.4 Results

According to the cardinality of the sets yielded by the mapping η_S^{-1} for each element of $T(S)$, such a final coalgebra may exist and may be defined. Hence, if we suppose that for every $S \in \mathbf{Set}$, and every $S' \in T(S)$, $\eta_S^{-1}(S')$ is a one element set, then given an endofunctor $\mathcal{H} = T(Out \times _)^{In}$, we can define a coalgebra (Γ, π) over $\mathcal{H}$ and show that it is final in $\mathbf{Cat}(\mathcal{H})$. But before, let us introduce some notions that will be useful for this purpose.

Definition 7 (Derivative dataflow). *Let x be a dataflow over a set A. The dataflow x'* **derivative** *of x is defined by: $\forall n \in \omega,\ x'(n) = x(n+1)$.*
For every $a \in A$, let us note $a.x$ the dataflow y defined by:

$$y(0) = a \quad \text{and} \quad \forall n \in \omega \setminus \{0\},\ y(n) = x(n-1)$$

Hence, $x = x(0).x'$.

Definition 8 (Derivative function). *Let T be a monad. Let In and Out be two sets denoting, respectively, the values in input and in output. Let $\mathcal{F} : In^\omega \to Out^\omega$ be a transfer function. For every input $i \in In$, we define the* **derivative function** *$\mathcal{F}_i : In^\omega \to Out^\omega$ for every $x \in In^\omega$ by $\mathcal{F}_i(x) = \mathcal{F}(i.x)'$*

The coalgebra (Γ, π) is then defined by $\Gamma = \{\mathcal{F} : In^\omega \to Out^\omega | \mathcal{F} \text{ is causal}\}$, and $\forall \mathcal{F} \in \Gamma, \forall i \in In, \pi(\mathcal{F})(i) = T(\{(\mathcal{F}[i], \mathcal{F}_i)\})$ where $\mathcal{F}[i] = \mathcal{F}(i.x)(0)$ for $x \in In^\omega$ chosen arbitrarily [5].

Theorem 1. *Let $\mathcal{H}$ be a the signature $T(Out \times _)^{In}$ such that for every $S \in \mathbf{Set}$, the mapping $\eta^{-1}_{Out \times S}$ for each $S' \in T(Out \times S)$ yields a one element set. Then, the coalgebra (Γ, π) is final in the category $\mathbf{Cat}(\mathcal{H})$. (see its proof in Appendix).*

This result can be extended to any monad T such that for every $S \in \mathbf{Set}$ and every $S' \in T(S)$, $\eta_S^{-1}(S')$ is of cardinality lesser than a cardinal κ. Indeed, let V be a set of cardinal κ. Let us consider the set of coalgebras over $\mathcal{H}$:

$$\mathcal{G} = \{(U, \gamma) | U \subseteq V \text{ and } \gamma : U \to \mathcal{H}(U)\}$$

Now, let us set:[6] $\Gamma = (\coprod_{(U,\gamma)\in\mathcal{G}} U)_{/\sim}$ and $\pi = (\coprod_{(U,\gamma)\in\mathcal{G}} \gamma)_{/\sim}$ where $\sim$ is the greatest bisimulation on $\coprod_{(U,\gamma)\in\mathcal{G}} U$.

Theorem 2. *With the conditions on cardinality, (Γ, π) is final in $\mathbf{Cat}(\mathcal{H})$ (see its proof in Appendix).*

[5] This makes sense because transfer functions are causal.
[6] In the literature, $\mathcal{G}$ is so-called a set of generators [25].

3 Conformance Testing for Components

In this section, we examine how we can test the conformance of an implementation of a component to its specification. In order to compare the behavior of the implementation to the specification, we need to consider both as components which have the same signature. However, the behavior of the implementation is unknown and can only be observed through its interface. We therefore need a conformance relation between what we can observe on the implementation and what the specification allows.

3.1 Conformance Relation

The specification *Spec* of a component is the formal description of its behavior given by a coalgebra over a signature $\mathcal{H} = T(Out \times _)^{In}$. On the contrary, its implementation *SUT* (for *System under Test*) is an executable component, which is considered as a black box [3, 13, 28]. We interact with the implementation through its interface, by providing inputs to stimulate it and observing its behavior through its outputs.

The theory of conformance testing defines the conformance of an implementation to a specification thanks to conformance relations. Several kinds of relations have been proposed. For instance, the relations of *testing equivalence* and *preorders* [23, 24] require the inclusion of trace sets. The relation *conf* [4] requires that the implementation behaves according to specification, but allows behaviors on which the specification puts no constrain. The relation *ioconf* [29, 30] is similar to *conf*, but distinguishes inputs from outputs. There are many other types of relations [12, 15, 20, 21].

In the following, we use the *ioconf* relation because it is the most suitable to our framework. For this we need to define some notions:

Definition 9. *Let $\mathcal{C} = (S, s_0, \alpha)$ be a component. Let $tr = \langle i_0|o_0, \ldots, i_n|o_n\rangle$ be a finite trace over $\mathcal{C}$* i.e. *an element of $Trace(\mathcal{C})$, and let s be a state of S. We have the two following definitions:*

- $(\mathcal{C} \text{ after } tr) = \{s' \mid \exists s_1, \ldots, s_n \in S,$
  $\forall j, 1 \leq j \leq n, (o_{j-1}, s_j) \in \eta^{-1}_{Out\times S}(\alpha(s_{j-1})(i_{(j-1)})),$
  and $(o_n, s') \in \eta^{-1}_{Out\times S}(\alpha(s_n)(i_n))\}$
  is the set of reachable states from the state s_0 after executing tr
- $Out_{\mathcal{C}}(s) = \bigcup_{i\in In} (\{o \mid \exists s' \in S, (o, s') \in \eta^{-1}_{Out\times S}(\alpha(s)(i))\})$
  is the set of the possible outputs in s.

  The set $Out_{\mathcal{C}}(s)$ can be extended to any set of states $S' \subseteq S$, we have :

$$Out_{\mathcal{C}}(S') = \bigcup_{s'\in S'} (Out_{\mathcal{C}}(s'))$$

These definitions allows us to define the *ioconf* relation in our framework:

Definition 10. *(ioconf) Let Spec and SUT be two components over the signature $T(Out \times _)^{In}$. The **ioconf** relation is defined as follows :*

$$SUT \text{ ioconf } Spec \Longleftrightarrow \begin{cases} \forall tr \in Trace(Spec), \\ Out_{SUT}(SUT \text{ after } tr) \subseteq Out_{Spec}(Spec \text{ after } tr) \end{cases}$$

3.2 Finite Computation Tree

In this section, we define the *finite computation tree* of a component, which captures all its finite computation paths:

Definition 11. *(Finite computation tree of component) Let (S, s_0, α) be a component over $T(Out \times _)^{In}$. The **finite computation tree** of depth n of $\mathcal{C}$, noted $FCT(\mathcal{C}, n)$ is the coalgebra $(S_{FCT}, s^0_{FCT}, \alpha_{FCT})$ defined by :*

- *S_{FCT} is the whole set of $\mathcal{C}-$*paths*. A $\mathcal{C}-$*path *is defined by two finite sequences of states and inputs $(s_0, \ldots, s_n)$ and $(i_0, \ldots, i_{n-1})$ such that for every $j, 1 \leq j \leq n, s_j \in \eta^{-1}_{Out \times S}(\alpha(s_{j-1})(i_{j-1}))_{|_2}$*
- *s^0_{FCT} is the initial $\mathcal{C}-$*path *$\langle s_0, () \rangle$*
- *α_{FCT} is the mapping which for every $\mathcal{C}-$*path *$\langle (s_0, \ldots, s_n), (i_0, \ldots, i_{n-1}) \rangle$ and every input $i \in In$ associates $T(\Gamma)$ where Γ is the set:*

$$\Gamma = \{(o, \langle (s_0, \ldots, s_n, s'), (i_0, \ldots, i_{n-1}, i) \rangle) \mid (o, s') \in \eta^{-1}_{Out \times S}(\alpha(s_n)(i))\}$$

In this definition, S_{FCT} is the set of the nodes of the tree. s^0_{FCT} is the root of the tree. Each node is represented by the unique $\mathcal{C}$-path $\langle (s_0, \ldots, s_n), (i_0, \ldots, i_{n-1}) \rangle$ which leads to it from the root:

$$s_0 \xrightarrow{i_0} s_1 \xrightarrow{i_1} \ldots \xrightarrow{i_{n-2}} s_{n-1} \xrightarrow{i_{n-1}} s_n$$

α_{FCT} gives, for each node p and for each input i, the set of nodes Γ that can be reached from p when input i is submitted to the component.

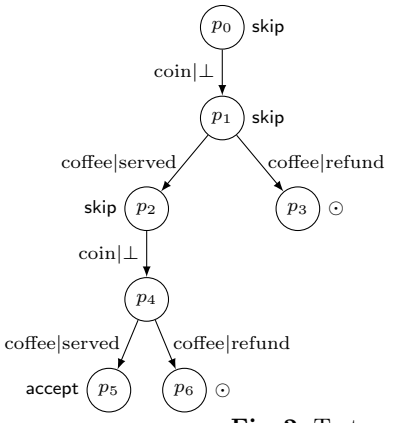
3.3 Test Purpose

In order to guide the test derivation process, test purposes can be used. A test purpose is a description of the part of the specification that we want to test and for which test cases are to be generated. In [6] test purposes are described independently of the model of the specification. On the contrary, we [10] prefer to describe test purposes by selecting the part of the specification that we want to explore. We therefore consider a test purpose as a tagged finite computation tree of the specification. The leaves of the FCT which correspond to paths that we want to test are tagged *accept*. All internal nodes on such paths are tagged *skip*, and all other nodes are tagged $\odot$.

Definition 12. *(Test Purpose) Let $FCT(\mathcal{C}, n)$ be the finite computation tree of depth n associated to a component $\mathcal{C}$. A **test purpose** TP for $\mathcal{C}$ is a mapping $TP : S_{FCT} \longrightarrow \{accept, skip, \odot\}$ such that:*

– *there exists a* $\mathcal{C}-$path $p \in S_{FCT}$ *such that* $TP(p) = accept$,
– *if* $TP(\langle(s_0, \ldots, s_n), (i_0, \ldots, i_{n-1})\rangle) = accept$, *then:*
  for every $j, 1 \leq j \leq n-1, TP(\langle(s_0, \ldots, s_j), (i_0, \ldots, i_{j-1})\rangle) = skip$
– $TP(\langle s_0, ()\rangle) = skip$
– *if* $TP(\langle(s_0, \ldots, s_n), (i_0, \ldots, i_{n-1})\rangle) = \odot$, *then:*
  $TP(\langle(s_0, \ldots, s_n, s'_{n+1}, \ldots, s'_m), (i_0, \ldots, i_{n-1}, i'_n, \ldots, i'_{m-1})\rangle) = \odot$
  for all $m > n$ *and for all* $(s'_j)_{n<j\leq m}$ *and* $(i'_k)_{n\leq k<m}$

Example 2. Figure 2 gives a test purpose TP on the finite computation tree of depth 4 of the coffee machine $\mathcal{M}$ whose specification is shown on Figure 1. This test purpose allows us to ignore the behaviors of $\mathcal{M}$ related to failure and repair and to concentrate on its interaction with a user. When the machine fails and the user is refunded, we reach node p_3 or p_6 which are tagged with $\odot$. This indicates that we are not interested in further behavior from these nodes. p_5 is tagged with accept because it is a leaf which corresponds to an expected behavior. All nodes leading from the root p_0 to this node are tagged with skip because they are valid prefixes of p_5.



$p_0 = \langle \text{STDBY}, () \rangle$
$p_1 = \langle (\text{STDBY}, \text{READY}), \text{coin} \rangle$
$p_2 = \langle (\text{STDBY}, \text{READY}, \text{STDBY}), (\text{coin}, \text{coffee}) \rangle$
$p_3 = \langle (\text{STDBY}, \text{READY}, \text{FAILED}), (\text{coin}, \text{coffee}) \rangle$
$p_4 = \langle (\text{STDBY}, \text{READY}, \text{STDBY}, \text{READY}), (\text{coin}, \text{coffee}, \text{coin}) \rangle$
$p_5 = \langle (\text{STDBY}, \text{READY}, \text{STDBY}, \text{READY}, \text{FAILED}), (\text{coin}, \text{coffee}, \text{coin}, \text{coffee}) \rangle$
$p_6 = \langle (\text{STDBY}, \text{READY}, \text{STDBY}, \text{READY}, \text{STDBY}), (\text{coin}, \text{coffee}, \text{coin}, \text{coffee}) \rangle$

Fig. 2. Test purpose of the coffee machine

In order to build a test purpose on a finite computation tree, we therefore choose the leaves of the tree which we accept as correct finite behaviors and we tag them with accept. We then tag every node which represents a prefix of an accepted behavior with skip. The other nodes, which lead to behaviors that we do not want to test, are tagged with $\odot$.
In the following, we use the notation TP to refer to an arbitrary test purpose.

4 Test generation guided by test purposes

Similarly to [10], we propose an approach for test cases selection according to a test purpose. In order to test the conformance of the SUT to the specification,

we start from the root of a finite computation tree, we choose a possible input i and submit it to the SUT. We observe the outputs o and compare them with the possible outputs in the finite computation tree. If the outputs do not match the specification, the verdict of the test is FAIL. Otherwise, if at least one of the nodes which can be reached with $i|o$ is tagged skip in the test purpose, the test goes on. If the nodes are tagged $\odot$, the test stops (further behavior is not of interest). Last, if one of the nodes is tagged accept, the test succeeds. A test case is therefore a sequence of input-output actions built

In this section, we describe the process which allows the construction of such a sequence. The main idea consists in choosing an input action according to the interactions with the SUT previously computed and the set of reachable states that can lead to accepting states of TP. Therefore, the reaction (output) received from the implementation is compared to the specified ones, and depending on the result of this comparison, our algorithm continues its computation, or stops by generating a verdict. Four verdicts are therefore distinguished : PASS, FAIL, INCONC and WeakPASS. Informally, PASS means that we reached an accept state and no observable difference between the specification and the implementation has been detected. FAIL means that the SUT had a behavior which is not allowed by the specification. INCONC means that no error has been detected but the behavior of the SUT went outside the test purpose, so we cannot conclude. Finally, WeakPASS means that the implementation behaved correctly but, due to the non-determinism of the specification, we are not sure to have reached an accept state (it may also be a $\odot$ one).

4.1 Preliminaries

In this section, we introduce some notations and definitions that will be used in describing our algorithm for generating conformance tests for components.

As mentioned above, a test case is considered as a sequence generated by TP interacting with SUT. This is denoted by $[ev_0, ev_2, \ldots, ev_n][Verdict]$, where for all $i \in [0, \ldots, n], ev_i = i|o$ is an input-output elementary sequence with $i \in In \cup \{\epsilon\}$ and $o \in Out \cup \{\epsilon\}$, and $Verdict \in \{FAIL, PASS, INCONC, WeakPASS\}$. We added the special symbol ϵ to the set of input actions In (*resp.* to the set of output actions Out) to denote the absence of output for a stimulation of the implementation (*resp.* a stimulation of SUT without an input). We note $stimobs(i|o)$ the output o received from SUT when stimulating it with input i.

In order to compute the set of reachable states that lead to *accept* states after a given input-output sequence, we define a current set of states denoted by CS that contains a subset of the states of the test purpose. It must be initialized at the beginning of our algorithm to the initial state of TP. Moreover, we introduce three functions : $Next(CS, ev), NextSkip(CS, ev)$ and $NextPass(CS, ev)$. These functions help exploring TP by selecting paths that lead to *accept* states. $Next(CS, ev)$ allows us to compute the set of directly reachable states from the current set states CS after executing ev. $NextSkip(CS, ev)$ computes the set of states belonging to $Next(CS, ev)$ from which it is possible to go to accepting states, and $NextPass(CS, ev)$ computes the set of states labelled by *accept*.

Definition 13. *Let* $TP : S_{FCT} \rightarrow \{accept, skip, \odot\}$ *be a test purpose for a component* $\mathcal{C}$*,* $ev = \langle i|o \rangle$ *an event, and* S' *a subset of* S_{FCT}*:*

- $Next(S', ev) = \bigcup_{s' \in S'} (\{s \mid (o, s) \in \eta^{-1}_{Out \times S_{FCT}}(\alpha_{FCT}(s')(i))\})$,
- $NextSkip(S', ev) = Next(S', ev) \bigcap TP(S')_{|skip}$,
- $NextPass(S', ev) = Next(S', ev) \bigcap TP(S')_{|accept}$.

with $TP(S')_{|\mathsf{tag}} = \{s' \in S' \mid TP(s') = \mathsf{tag}\}$

4.2 Inferences rules

We our test case generation algorithm as a set of inferences rules. Each rule states that under certain conditions on the next observation of output action from SUT or the next stimulation of SUT by an input action, the algorithm either performs an exploration of other states of TP, or stops by generating a verdict.

We structure these rules as $\frac{CS}{Results}$ $cond(ev)$, where CS is a set of current states, *Results* is either a set of current states or a verdict, and $cond(ev)$ is a set of conditions including $stimobs(ev)$. Each rule must be read as follows : *Given the current set of states CS, if* $cond(ev)$ *is verified, then the algorithm may achieve a step of execution, with ev as input-output elementary sequence.*

Our algorithm can be seen as an exploration of the finite computation tree starting from the initial state. It switches between sending stimuli to the implementation and waiting for output of the implementation according to the inference rules as long as a verdict is not reached. We distinguish two kinds of inference rules : *exploring* rules and *diagnosis* rules. The first kind, is applied to pursue the computation of the sequence as long as *Result* is a set of states. The second kind leads to a verdict and stops the algorithm.

Rule 0 : Initialization rule[7]: $\frac{}{\{s^0_{FCT}\}}$

Rule 1 : Generation of the verdict FAIL: the output from the SUT is not expected with regards to the specification.

$$\frac{CS}{FAIL}\ stimobs(ev),\ Next(CS, ev) = \emptyset$$

Rule 2 : Generation of the verdict FAIL : the emission from the SUT is not expected with regards to the specification.

$$\frac{CS}{FAIL}\ stimobs(ev),\ Next(CS, ev) = \emptyset$$

[7] This rule is involved only once when starting the algorithm.

Rule 3 : Generation of the verdict INCONC : the emission from the SUT is specified but not compatible with the test purpose.

$$\frac{CS}{INCONC} stimobs(ev), \begin{cases} Next(CS, ev) \neq \emptyset, \\ NextSkip(CS, ev) = NextPass(CS, ev) = \emptyset \end{cases}$$

Rule 4 : Generation of the verdict PASS : all next states directly reachable from the set of current set are *accept* ones.

$$\frac{CS}{PASS}\ stimobs(ev),\ NextPass(CS, ev) = Next(CS, ev),\ Next(CS, ev) \neq \emptyset$$

Rule 5 : Generation of the verdict WeakPASS : some of the next states are labelled by *accept*, but not all of them.

$$\frac{CS}{WeakPASS} stimobs(ev), \begin{cases} NextPass(CS, ev) \subset Next(CS, ev), \\ NextPass(CS, ev) \neq \emptyset \end{cases}$$

We should now note that each of these rules except rule 0 can be used in several ways according to the form of ev. When $ev = \epsilon|o$, o is produced spontaneously by SUT. When $ev = i|\epsilon$, the stimulation of SUT with i does not produce any output. Finally, when $ev = i|o$, o is produced by SUT when it is stimulated with i. These possibilities for ev therefore give rise to a generic algorithm that can be applied to a wide variety of state-based systems. This means that an appropriate choice of the monad T and both input In and output Out sets, allows us to obtain models as defined in [6, 7, 10, 16, 29].

4.3 Properties

A test case informs us about the conformance of the implementation to its specification. This means that the non-existence of a FAIL verdict leads to a conformance, and that any non-conformance should be detected by a test case ending by a FAIL verdict. In order to study the coherence between the notion of conformance of an implementation under test and its specification, and the notion of test case generated by our algorithm, we denote by $\mathbb{CS}$ and $\mathbb{EV}$ respectively the whole set of current state sets and the whole set of input-output elementary sequences used during the application of the set of inference rules on an implementation SUT according to a test purpose TP. We then introduce a transition system whose states are the sets of current states and four special states labelled by the verdicts. Two states are linked by a transition labelled by an input-output elementary sequence. This transition system is formally defined as follows :

Definition 14. *Let TP be a test purpose for a specification Spec, let SUT be an implementation, let $\mathbb{CS}$ be the whole set of current state sets and let $\mathbb{EV}$ be the whole set of input-output elementary sequences. Then,* ***the execution of the test generation algorithm*** *on SUT according to TP denoted by $TS(TP, SUT)$ (see its explanation in Section 4.2) is the coalgebra (S_{TS}, α_{TS}) over the signature $(_)^{\mathbb{EV}}$ defined by :*

– $S_{TS} = \mathbb{CS} \cup \mathbb{V}\text{erdict}$ *where* $\mathbb{V}\text{erdict}$ *is the set whose elements are FAIL, PASS, INCONC and WeakPASS,*
– α_{TS} *is the mapping which for every* $CS \in \mathbb{CS}$ *and for every* $ev \in \mathbb{EV}$ *is defined as follows :*

$$\alpha_{TS}(CS)(ev) = \begin{cases} Next(CS, ev) & \text{if } NextSkip(CS, ev) \neq \emptyset, NextPass(CS, ev) = \emptyset \\ FAIL & \text{if } Next(CS, ev) = \emptyset \\ INCONC & \text{if } NextSkip(CS, ev) = NextPass(CS, ev) = \emptyset \\ & \quad \text{and } Next(CS, ev) \neq \emptyset \\ PASS & \text{if } Next(CS, ev) = NextPass(CS, ev) \\ & \quad \text{and } Next(CS, ev) \neq \emptyset \\ WeakPASS & \text{if } NextPass(CS, ev) \subsetneq Next(CS, ev) \\ & \quad \text{and } NextPASS(CS, ev) \neq \emptyset \end{cases}$$

With this definition, test cases are sets of possible traces which can be observed during an execution of $TS(TP, SUT)$, and lead to a verdict state.

Definition 15. *Let* $TS(TP, SUT) = (S_{TS}, \alpha_{TS})$ *be the execution of the test generation algorithm on* SUT *according to* TP*. A* ***test case*** *for* TP *is a sequence* $[ev_0, \ldots, ev_n][Verdict]$ *for which there is a sequence of states* $s_0, \ldots, s_n \in \mathbb{CS}$ *with* $\forall j, 0 \leq j < n, s_{j+1} = \alpha_{TS}(s_j)(ev_j)$*, and there is a verdict state* $Verdict \in \mathbb{V}\text{erdict}$ *such that* $Verdict = \alpha_{TS}(s_n)(ev_n)$*. We note* $st(TP, SUT)$ *the set of all possible test cases for* TP*.*

We can now introduce the notation:

$vdt(TP, SUT) = \{Verdict \mid \exists ev_0, \ldots, ev_n, [ev_0, \ldots, ev_n | Verdict] \in st(TP, SUT)\}$

Theorem 3. *(Correctness and completeness) For any specification Spec and any SUT:*

– ***Correctness:*** *If* SUT *conforms to Spec, for any test purpose* TP*,* $FAIL \notin vdt(TP, SUT)$*.*
– ***Completeness:*** *If* SUT *does not conform to Spec, there exists a test purpose* TP *such that* $FAIL \in vdt(TP, SUT)$*.*

(See its proof in Appendix)

5 Conclusion

In this paper, we have presented a coalgebraic model, a conformance relation between implementations and specifications, and a test generation algorithm for component based systems. This work relies on previous work by Barbosa [1, 19] for defining software components as coalgebras, and defines a framework which encompasses Mealy machines, labeled transition systems and input-output labeled transition systems. It still has to be applied to systems with data.

A conformance testing theory has been generalized to our framework. It considers both the specification and the implementation models as components, and

defines a generic version of the relation of conformance testing proposed in [29]. The main idea is to replace the specification by a finite computation tree whose set of paths denotes the set of all computations allowed by the specification. We have used test purposes to narrow the computation tree by pruning the parts which are not of interest for testing. An algorithm to test the conformance of an implementation to a specification according to a test purpose has been proposed. It is based on the exploration of the computation tree according to a set of rules. This exploration stops when a leaf of the test purpose is reached (the test passes), when the behavior of the SUT leaves the test purpose (the test is unconclusive), or when the behavior of the SUT leaves the specification (the test fails). Moreover, we have proved that this algorithm is correct and complete with regard to the conformance relation *ioconf*.

The ability of this framework to model and generate tests for various kinds of components is a step toward the testing of heterogeneous systems, made from components specified using different formalisms. This requires the definition of integration operators to combine the behavior of components. It should allow us to check wether an implementation made of conforming components combined with integration operators is conform to its specification.

References

1. L.S Barbosa. Towards a calculus of state-based software components. *Journal of Universal Computer Science*, 9(8):891–909, August 2003.
2. M. Barr and C. Wells, editors. *Category theory for computing science, 2nd ed.* Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1995.
3. G. Bernot. Testing against formal specifications: A theoretical view. In *TAPSOFT '91: Proceedings of the International Joint Conference on Theory and Practice of Software Development, Volume 2: Advances in Distributed Computing (ADC) and Colloquium on Combining Paradigms for Software Developmemnt (CCPSD)*, pages 99–119, London, UK, 1991. Springer-Verlag.
4. E. Brinksma. A theory for the derivation of tests. *Protocol Specification, Testing, and Verification (PSTV VIII)*, pages 63–74, 1988.
5. S. Brookes and A. W. Roscoe. An Improved Failures Model for Communicating Processes. *NSF-SERC Seminar on Concurrency, Pittsburgh, July 1984. Springer LNCS 197.*, pages 281–305, 1985.
6. T. Jéron C. Jard. Tgv : theory, principles and algorithms. *International Journal on Softaware Tools for Technology Transfer*, 7(4):297–315, August 2000.
7. S. Eilenberg. *Automata, Languages and Machines*, volume C. Academic Press, New York, 1978.
8. J. L. Fiadeiro. *Categories for Software Engineering.* SpringerVerlag, 2004.
9. L. Frantzen, J. Tretmans, and T.A.C. Willemse. A Symbolic Framework for Model-Based Testing. In K. Havelund, M. Núñez, G. Rosu, and B. Wolff, editors, *Formal Approaches to Software Testing and Runtime Verification – FATES/RV 2006*, number 4262 in Lecture Notes in Computer Science, pages 40–54. Springer, 2006.
10. C. Gaston, P. Le Gall, N. Rapin, and A. Touil. Symbolic execution techniques for test purpose definition. In *TestCom*, pages 1–18, 2006.

11. J. Rutten H. Hvid Hansen, D. Costa. Synthesis of mealy machines using derivativesesis of mealy machines. *Proceedings of the Eighth Workshop on Coalgebraic Methods in Computer Science (CMCS 2006), Eighth Workshop on Coalgebraic Methods in Computer Science*, 164:27–45, octobre 2006.
12. C.A.R. Hoare. Communicating sequential processes. *cacm*, 21(8):666–677, 1978.
13. Project 1.21.54. ISO. Formal methods in conformance testing, working draft. In *Selected proceedings of the IFIP TC6 9th international workshop on Testing of communicating systems*, February 1995.
14. B. Jeannet, T. Jéron, V. Rusu, and E. Zinovieva. Symbolic test selection based on approximate analysis. In N. Halbwachs and L.D. Zuck, editors, *TACAS*, volume 3440 of *Lecture Notes in Computer Science*, pages 349–364. Springer, 2005.
15. R. Langerak. A testing theory for lotos using deadlock detection. In *Proceedings of the IFIP WG6.1 Ninth International Symposium on Protocol Specification, Testing and Verification IX*, pages 87–98, Amsterdam, The Netherlands, The Netherlands, 1990. North-Holland Publishing Co.
16. D. Lee and M. Yannakakis. Principles and methods of testing finite state machines—a survey. *Proceedings of the IEEE*, 84(8), August 1996.
17. S. Mac Lane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer Verlag, New York, Heidelberg, Berlin, 1971.
18. G. H. Mealy. A method for synthesizing sequentiel circuits. *Bell Systems Techn. Jour.*, 1955.
19. S. Meng and L.S. Barbosa. Components as coalgebras: the refinement dimension. *Theor. Comput. Sci.*, 351(2):276–294, 2006.
20. R. Milner. A calculus of communicating systems. *Springer-Verlag New York, Inc, secaucus, NG, USA*, 1982.
21. R. Milner. *Communication and concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.
22. E. Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.
23. R. De Nicola and M. C. B. Hennessy. Testing equivalences for processes. *tcs*, 34(1–2):83–133, nov 1984.
24. I. Phillips. Refusal testing. *Theor. Comput. Sci.*, 50(3):241–284, 1987.
25. J. Rutten. Universal coalgebra: a theory of systems. *Theor. Comput. Sci.*, 249(1):3–80, 2000.
26. J. Rutten. Algebraic specification and coalgebraic synthesis of mealy machines. *Technical Report SEN-R0514, Centrum voor Wiskunde en Informatica (CWI)*, 2005.
27. E.D. Sontag. *Mathematical control theory: deterministic finite dimensional systems (2nd ed.)*. Springer-Verlag New York, Inc., New York, NY, USA, 1998.
28. J. Tretmans. A formal approach to conformance testing. In *Proceedings of the IFIP TC6/WG6.1 Sixth International Workshop on Protocol Test systems VI*, pages 257–276, Amsterdam, The Netherlands, The Netherlands, 1994. North-Holland Publishing Co.
29. J. Tretmans. Conformance testing with labelled transition systems : Implementation relations and test generation. *Computer networkss and ISDN systems*, 29(1):49–79, 1996.
30. J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools*, 17(3):103–120, 1996.

Appendix

– **Proof of theorem 1 (page 8).** For every coalgebra (S, α), let us define $!_\alpha : S \to \Gamma$ which for every $s \in S$, associates the transfer function $!_\alpha(s) : In^\omega \to Out^\omega$ defined for every $s \in S$, every $x \in In^\omega$ and every $k \in \omega$ as follows. Let $s_0, \ldots, s_k \in S$ the sequence of state such that $s_0 = s$ and:

$$\forall j, 1 \leq j \leq k-1, s_j = \eta^{-1}_{Out \times S}(\alpha(s_{j-1})(x(j-1)))_{|_2}$$

This sequence exists and by hypothesis on cardinality of $\eta^{-1}_{Out \times S}$, is unique. Then, let us set:

$$!_\alpha(s)(x)(k) = \eta^{-1}_{Out \times S}(\alpha(s_k)(x(k)))_{|_1}$$

It is not very difficult to check that $!_\alpha(s)$ is causal, and that $!_\alpha$ defined in this way is a homomorphism, which is further unique.

– **Proof of theorem 2 (page 8).**
For every coalgebra (S, α), there exists by definition a coalgebra (U, γ) in $\mathcal{G}$ such that (S, α) and (U, γ) are isomorphic. Obviously, two isomorphic coalgebras are bisimilar. Therefore, we can define the homomorphism $!_\alpha$ which to (S, γ) associates the unique element $[(U, \gamma)]$ in Γ where $[(U, \gamma)]$ is the equivalence class of (U, γ) for $\sim$.

– **Proof of theorem 3 (page 15).**
Proof of the correctness : Let $Spec = (S, s_0, \alpha)$ be a specification over $\mathcal{H} = T(Out \times _)^{In}$ and $FCT = (S_{FCT}, s^0_{FCT}, \alpha_{FCT})$ be its finite computation tree. Let us prove the correctness using the contraposition principle. This means that to prove :

if SUT conforms to Spec, for any test purpose TP, FAIL $\notin vdt(TP, SUT)$.

we have to prove :

if there exists a test purpose TP such that FAIL $\in vdt(TP, SUT)$,

then SUT does not conform to Spec.

More precisely, according to the definition of *ioconf*, we have to prove that : there exists a finite trace $tr \in Trace(FCT)$ such that $Out_{SUT}(SUT \text{ after } tr) \not\subseteq Out_{FCT}(FCTc \text{ after } tr)$. This is proved by the following proposition :

Proposition 1. *If there exists a test purpose TP such that $[ev_0, \ldots, ev_n | FAIL] \in st(TP, SUT)$, then :*

1. $ev_0 \ldots ev_{n-1} \in Trace(FCT)$.
2. $ev_n \in Out_{SUT}(SUT \text{ after } (ev_0 \ldots ev_{n-1}))$.
3. $ev_0 \ldots ev_n \notin Trace(FCT)$.
4. $ev_0 \ldots ev_n \in Trace(SUT)$.

Proof of (1).
In order to show that the sequence $ev_0 \dots ev_{n-1} \in Trace(FCT)$, we are going to reason on the way of computation of this sequence by using the inference rules. First of all, let $TS(TP, SUT)$ be the execution of the test generation algorithm and $st(TP, SUT)$ be the set of generated test cases. Since $[ev_0, \dots, ev_n|FAIL] \in st(TP, SUT)$, then there exists for every $j, 0 \leq j < n, S_j \in \mathbb{CS}$ such that $S_0 = \{s^0_{TS}\}, S_{j+1} = \alpha_{TS}(S_j)(ev_j)$ and $FAIL = \alpha_{TS}(S_n)(ev_n)$. Hence, for every $j, 0 \leq j < n, S_{j+1}$ which equals to $Next(S_j, ev_j)$ is not empty by Definition 14. Hence, by Definition 13, for every $j, 0 \leq j < n$, every state belonging into S_{j+1} is a state of FCT. This means that for every $j, 0 \leq j < n$, every state $s \in S_j$ is related to a state $s' \in S_{j+1}$ by ev_j. Consequently, the sequence $ev_0 \dots ev_j \dots ev_{n-1} \in Trace(FCT)$.

Proof of (2).
It is obvious because $[ev_0 \dots ev_n|FAIL] \in st(TP, SUT)$.

Proof of (3).
We have above proved that $ev_0 \dots ev_{n-1} \in Trace(FCT)$ and $S_n \neq \emptyset$. We have that $[ev_0 \dots ev_n|FAIL] \in st(TP, SUT)$ *i.e.* applying ev_n have to lead to a FAIL verdict. This means that $\alpha_{TS}(S_n)(ev_n) = FAIL$. Hence by Definition 14, $Next(S_n, ev_n)$ have to be empty. But we know that $Next(S_n, ev_n) \subseteq S_{FCT}$. Hence, $ev_0 \dot{e} v_{n-1} ev_n$ does not belong to $Trace(FCT)$.

Proof of (4).
It is obvious because $[ev_0 \dots ev_n|FAIL] \in st(TP, SUT)$.

Proof of the completeness : Let $Spec = (S, s_0, \alpha)$ be a specification over a signature $\mathcal{H} = T(Out \times _)^{In}$ and $FCT = (S_{FCT}, s^0_{FCT}, \alpha_{FCT})$ be its finite computation tree. Let us prove that the completeness holds. For this, let us assume that SUT does not conform to $Spec$ and let us prove that there exists a test purpose TP such that there exists $[ev_0, \dots, ev_n|FAIL] \in st(TP, SUT)$.
First of all, SUT does not conform to $Spec$. According to the definition of *ioconf*, there exists a trace $tr = ev_0 \dots ev_{n-1} \in Trace(FCT)$ such that $Out_{SUT}(SUT \text{ after } tr) \nsubseteq Out_{FCT}(FCT \text{ after } tr)$ *i.e.* there exists $ev_n = i|o$ such that $o \in Out_{SUT}(SUT \text{ after } tr)$, $ev_0 \dots ev_{n-1} ev_n \in Trace(SUT)$ and $ev_0 \dots ev_{n-1} ev_n \notin Trace(FCT)$.
Now, let us denote by TP a test purpose of FCT such that there exists $ev'_n \in \mathbb{EV}$ and a state $s \in S_{FTC}$ such that $s \in (FCT \text{ after } ev_0 \dots ev_{n-1} ev'_n)$ and $TP(s) = accept$ *i.e.* $ev_0 \dots ev_{n-1} ev'_n$ forms a path of TP. Let us prove that there exists $[ev_0 \dots ev_{n-1} ev_n|FAIL] \in st(TP, SUT)$. For this, it is enough to show that there exists $(S_j)_{0 \leq j \leq n}$ such that for every $j, 0 \leq j < n, S_{j+1} = \alpha_{TS}(S_j)(ev_j) \in \mathbb{CS}$ and $FAIL = \alpha_{TS}(S_n)(ev_n)$.
We have that $ev_0 \dots ev_{n-1} \in Trace(FCT)$, then, for every $j, 0 \leq j \leq n, S_j$ exists because for every $j, 1 \leq j \leq n, \alpha_{TS}(S_j)(ev_j) = Next(S_j, ev_j)$ and $S_0 = \{s^0_{FCT}\}$. Thus, what remains is to prove that there is a verdict state

FAIL such that $FAIL = \alpha_{TS}(S_n)(ev_n)$. By hypothesis, $ev_0 \dots ev_{n-1} ev_n \notin Trace(FCT)$ and $ev_0 \dots ev_{n-1} ev_n \in Trace(SUT)$, hence $Next(S_n, ev_n) = \emptyset$, and consequently $\alpha_{TS}(S_n)(ev_n) = FAIL$.