Testing of Abstract Components

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

[1] École Centrale Paris
Laboratoire de Mathématiques Appliquées aux Systèmes (MAS)
Grande Voie des Vignes F-92295 Châtenay-Malabry
{marc.aiguier,bilal.kanso}@ecp.fr
[2] SUPELEC Systems Sciences (E3S) - Computer Science Department
3 rue Joliot-Curie F-91192 Gif-sur-Yvette cedex
{frederic.boulanger,assia.touil}@supelec.fr

Abstract. In this paper, we present a conformance testing theory for Barbosa's abstract components. We do so by defining a trace model for components from causal transfer functions which operate on data flows at discrete instants. This allows us to define a test selection strategy based on test purposes which are defined as subtrees of the execution tree built from the component traces. Moreover, we show in this paper that Barbosa's definition of components is abstract enough to subsume a large family of state-based formalisms such as Mealy machines, Labeled Transition Systems and Input/Output Labeled Transition Systems. Hence, the conformance theory presented here is a generalization of the standard theories defined for different state-based formalisms and is a key step toward a theory of the test of heterogeneous systems.

Keywords: Component based system, Coalgebra, Monad, Trace semantics, Transfert function, Conformance testing, Test purpose.

1 Introduction

The design of complex software systems relies on the hierarchical composition of subsystems which may be modeled using different formalisms [12]. These subsystems can be considered as state-based components whose behavior can be observed at their interface. In order to model such components in an abstract way, we use a definition introduced by Barbosa in [1,19]. The interest of this definition is twofold: first, it can be used to describe differents kinds of state-based formalisms, and second, it extends Mealy machines, which, following Rutten's work [11], allows us to define a trace model over components using causal transfer functions. Barbosa defines a component as a coalgebra over the endofunctor $\mathcal{H} = T(Out \times \_)^{In}$ where T is a monad[1], and In and Out are two sets of elements which denote respectively inputs and outputs of the component. Hence, Barbosa's definition of a component is an extension of Mealy automata [18], which are efficient to specify the behavior of components deterministically. The

[1] The definitions and notations used in this paper are recalled in Section 2.

role of the T monad is to take into account in a generic way various computation structures such as non-determinism, partiality, etc [21]. Therefore, Barbosa's definition of a component allows us to model components independently of any computation structure but also independently of the state-based formalisms classically used to specify software components. Indeed we show in this paper that state-based formalisms such as Mealy automata, Labeled Transition Systems and Input-Output Symbolic Transitions [8,9] can be embedded into Barbosa's definition by a suitable choice for the monad T. Moreover, this way of modeling the behavior of components allows us, following Rutten's works [11], to define a trace model over components by causal transfer functions. Such functions are 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 the time which is considered here as discrete.

Indeed, defining a trace model from causal functions allows us, first to show the existence of a final coalgebra in the category of coalgebras over a signature $T(Out \times \_)^{In}$ under some sufficient conditions on the monad T, and second, to define a conformance testing theory for components, which is the main contribution of this paper. Final coalgebras are important because their existence is the key of *co-induction*, a powerful reasoning principle. Following some previous works by some authors of this paper [9], we define test purposes as particular subtrees of the execution tree built from our trace model for components. Then, we define an algorithm for generating test cases from a test purpose. Like in [9], this algorithm is given as a set of inference rules. Each rule is dedicated to the handling of an observation of the system under test (SUT) or of a stimulation sent by the test case to the SUT.

The paper is structured as follows: Section 2 recalls the basic notions of the categorical theory of coalgebras and monads that are used in this paper. Then, Section 3 recalls Barbosa's definition of components and introduces our trace model from causal transfer functions. The formalization of components as coalgebras allows us to extend standard results connected to the definition of a terminal component. Section 4 presents our conformance testing theory for components, and Section 5 gives the inference rules for generating test cases.

2 Preliminaries

This paper relies on many terms and notations from the categorical theory of coalgebras and monads. We briefly introduce them here, but interested readers may refer to textbooks such as [2,7,17].

2.1 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 $\varepsilon_Y \circ F(f) = G(f) \circ \varepsilon_X$.

2.2 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 the sets of states, inputs and outputs.

2.3 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:

$$\begin{tikzcd}
F(X) \arrow[r, "F(f)"] \arrow[d, "\alpha"'] & F(Y) \arrow[d, "\beta"'] \\
X \arrow[r, "f"'] & Y
\end{tikzcd}
\qquad
\begin{tikzcd}
Z \arrow[r, "g"] \arrow[d, "\delta"'] & U \arrow[d, "\gamma"'] \\
F(Z) \arrow[r, "F(g)"'] & F(U)
\end{tikzcd}$$

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).

2.4 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 $\mu \circ T\mu = \mu \circ \mu T$:

$$\begin{tikzcd}
T^2 \arrow[dr, "\mu"'] & T \arrow[l, "T\eta"'] \arrow[r, "\eta T"] \arrow[d, "\mathsf{id}_{\mathbb{C}}"] & T^2 \arrow[dl, "\mu"] \\
& T &
\end{tikzcd}
\qquad
\begin{tikzcd}
T^3 \arrow[r, "T\mu"] \arrow[d, "\mu T"'] & T^2 \arrow[d, "\mu"] \\
T^2 \arrow[r, "\mu"'] & T
\end{tikzcd}$$

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 flattening two levels of structure can be made either by flattening the outer (with $\mu_{T(X)}$) or the inner (with $T(\mu_X)$) structure first.

Let us consider a monad built on the powerset functor $\mathcal{P} : \mathbf{Set} \to \mathbf{Set}$. We use it to model non-deterministic state machines by replacing the target state of a transition by a set of possible states. The component $\eta_S : S \to \mathcal{P}(S)$ of the unit of this monad has to build a set of states from a state. We can 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 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 choose $\mu_S : \{s_1 \ldots s_i \ldots\} \mapsto \cup s_i$.

Moreover, monads have also been used to represent many computation situations such as partiality, side-effects, exceptions, *etc* [21].

3 Transfer Functions and Components

In this section, we 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 3.2, the interest of Barbosa's definition of components is that it is abstract enough to unify in a same framework a large family of formalisms classically used to specify state-based systems, such as Mealy machines, Labelled Transition Systems (LTS), Input-Output Labelled Transition Systems (IOLTS), etc. Similarly to Rutten's works in [11], we denote the behavior of a component by a transfer function.

3.1 Transfer Function

In the following, we 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.*

Transfer functions, which we use to describe the observable behavior of components, can be seen as dataflow transformers satisfying the causality condition in a standard framework [24], that is the output data at index n only depends on input data at indexes $0, \ldots, n$.

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:*

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

3.2 Components

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

Example 1. We illustrate the notions and results previously mentioned with the simple example of a coffee machine $\mathcal{M}$ modeled by 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, it goes into the READY state. Then, when the user presses the "coffee" button, it 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[2] $\mathcal{P}_f(Out \times \_)^{In}$. The state space is

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

$S = \{\text{STDBY}, \text{READY}, \text{FAILED}\}$ and $s_0 = \text{STDBY}$. The sets of inputs and outputs are $In = \{\text{coin}, \text{coffee}, \text{repair}\}$ and $Out = \{\perp, \text{served}, \text{refund}\}$. Finally, the transition function $\alpha : S \longrightarrow \mathcal{P}_f\big(\{\perp, \text{served}, \text{refund}\} \times S\big)^{\{\text{coin},\text{coffee},\text{repair}\}}$ is defined as follows:

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

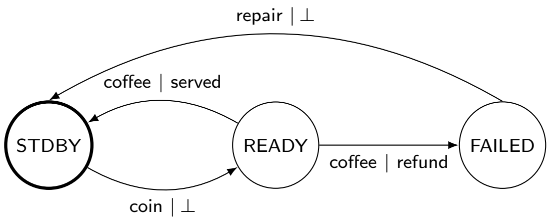


Fig. 1. Coffee machine

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 components*** *over $\mathcal{H}$.*

Using Definition 3 for components, we can 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, when T is the identity functor $\mathcal{I}d$, the resulting component is a Mealy machine. A Labelled Transition System is obtained by choosing $Out = \{\}$ and $In = Act$, a set of symbols standing for actions names, and the powerset functor $\mathcal{P}$ for T. Finally, with the powerset monad $\mathcal{P}$ for T, and with the additional property on the transition function $\alpha : S \longrightarrow \mathcal{P}(Out \times S)^{In}$:

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

we obtain an *IOLTS* (input and output are mutually exclusive).

3.3 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 $T : S \mapsto S \cup \{\perp\}$, η_S^{-1} is the mapping that associates to $s \in S$ the singleton $\{s\}$ and the emptyset for $\perp$. 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 be different of the structure of $Out \times S$. The mapping $\eta^{-1}_{Out \times S}$ maps back to this structure.

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 $\alpha(s)(i)$.

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 finite sequences 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, $Trace(\mathcal{C}) = \bigcup\limits_{\mathcal{F}_{s_0}} \bigcup\limits_{n \in \mathbb{N}} \mathcal{F}_{{s_0}_{|n}}$ defines the whole set of finite traces over $\mathcal{C}$.

4 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 over a 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.

4.1 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 [3,25]. 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* [22,23] require the inclusion of trace sets. The relation *conf* [4] requires that the implementation behaves according to a specification, but allows behaviors on which the specification puts no constrain. The relation *ioconf* [26] is similar to *conf*, but distinguishes inputs from outputs. There are many other types of relations [15,20].

conf and *ioconf* have received most attention by the community of formal testing because they have shown their suitability for conformance testing. Since we are dealing with components with input and output, we choose *ioconf* and extend it to fit our framework. There are several extentions to *ioconf* according to both the underlying type of transition system and the aspect considered to be tested [8,9,13,5]. Recently, a denotational version of *ioconf* [27] was redefined in the Unifying Theories of Programming (UTP) [10].

Definition 7. *Let $\mathcal{C} = (S, s_0, \alpha)$ be a component. Let $tr = \langle i_0|o_0, \dots, i_n|o_n\rangle$ 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:*

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

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

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

Definition 8. *(ioconf) Let Spec and SUT be two components over the signature $T(Out \times \_)^{In}$. The* ***ioconf*** *relation is defined as follows :*

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

We should note here that our *ioconf* definition covers all possible assumptions that must classically be made in conformance testing practice. For instance, it is always assumed that implementations are input enabled, that is, at any state, the implementation must produce an answer for all outputs. This assumption can naturally be expressed in our framework by considering the transition function α as total function.

4.2 Finite Computation Tree

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

Definition 9. *(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* $j, 1 \leq j \leq n, s_j \in \eta^{-1}_{Out \times S}\big(\alpha(s_{j-1})(i_{j-1})\big)_{|_2}$
- 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:*

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

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 the input i is submitted to the component.

4.3 Test Purpose

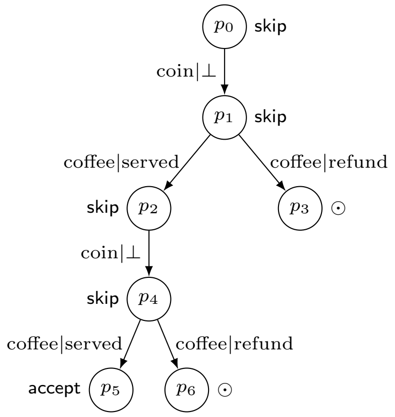
In order to guide the test derivation process, test purposes can be used. A test purpose is a description of the part of the specification that we want to test and for which test cases are to be generated. In [6] test purposes are described independently of the model of the specification. On the contrary, following [9], we 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 10. *(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* $TP : S_{FCT} \longrightarrow \{\mathsf{accept}, \mathsf{skip}, \odot\}$ *such that:*

- *there exists a* $\mathcal{C}-$path $p \in S_{FCT}$ *such that* $TP(p) = \mathsf{accept}$
- *if* $TP(\langle (s_0, \ldots, s_n), (i_0, \ldots, i_{n-1})\rangle) = \mathsf{accept}$, *then:*
  for every $j, 1 \leq j \leq n-1, TP(\langle (s_0, \ldots, s_j), (i_0, \ldots, i_{j-1})\rangle) = \mathsf{skip}$

- $TP(\langle s_0, ()\rangle) = \mathsf{skip}$
- *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 $\odot$.

5 Test Generation Guided by Test Purposes

Similarly to [9], 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 test purpose, 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 $\odot$, further behavior is not of interest, so the test is inconclusive (INCONC verdict). If one of the nodes is tagged accept, the test succeeds (PASS verdict). It may happen, due to the non-determinism of the specification, that the implementation behaved correctly, but we cannot determine if we reached an accept state or an $\odot$ state. This leads to a WeakPASS verdict.

5.1 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 a sequence generated by a test purpose TP interacting with SUT. This is denoted by $[ev_0, ev_2, \dots, ev_n][Verdict]$, where for all $i \in [0, \dots, n], ev_i = i|o$ is an elementary input-output 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 input and output actions to denote a stimulation of SUT without input and the absence of output for a stimulation. We note $stimobs(i|o)$ the output o 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 is initialized to the initial state of TP. We also introduce three functions to help exploring TP by selecting paths that lead to *accept* states. $Next(CS, ev)$ is the set of directly reachable states from the current set of states CS after executing ev. $NextSkip(CS, ev)$ is the set of states in $Next(CS, ev)$ from which it is possible to reach accepting states, and $NextPass(CS, ev)$ is the set of states in $Next(CS, ev)$ which are labelled by *accept*.

Definition 11. *Let $TP : S_{FCT} \to \{accept, skip, \odot\}$ be a test purpose for a component $\mathcal{C}$, $ev = \langle i|o \rangle$ an event, and S' a subset of S_{FCT}:*

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

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

5.2 Inferences Rules

We define 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[3]: $\frac{}{\{s^0_{FCT}\}}$

Rule 1 : Exploration of other states : the emission o after a stimulation by i on the SUT is compatible with the test purpose but no accept is reached.

$$\frac{CS}{Next(CS,ev)}\ stimobs(ev),\ NextSkip(CS,ev) \neq \emptyset$$

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

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

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 ([6,9,16]) by choosing the appropriate monad T and input and output sets.

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

5.3 Properties

In order to state that, according to our algorithm, the non-existence of a FAIL verdict leads to conformance (correctness) and that any non-conformance is detected by a test case ending by a FAIL verdict (completeness), 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 12. *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 $TS(TP, SUT)$ 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 13. *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 1. *(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* $FAIL \in vdt(TP, SUT)$.

6 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 works done in [1,19] for defining software components as coalgebras, and in [9] for defining our test generation algorithm.

The formalism used in this paper to specify both the specification and system the under test is abstract enough to subsume most state-based formalisms. Hence, the conformance theory defined here over this formalism is *de facto* a generalization of standard theories found for different state-based formalisms.

The ability of this framework to model and generate tests for components is a first step toward the testing of complex (software) systems, made from a huge number of components that interact altogether. This will require the definition of integration operators to combine the behavior of components. It should then allow us to check whether an implementation made of conforming components combined with integration operators is conform to its specification.

In order to fit the required format of the paper, we omitted some details (detailed explanations, theorems of existence of final coalgebras, proofs of all theorems) which are available in an extended version of this paper [14].

References

1. Barbosa, L.S.: Towards a calculus of state-based software components. Journal of Universal Computer Science 9(8), 891–909 (2003)
2. Barr, M., Wells, C. (eds.): Category theory for computing science, 2nd edn. Prentice Hall International (UK) Ltd., Hertfordshire (1995)
3. Bernot, G.: Testing against formal specifications: A theoretical view. In: TAPSOFT 1991: Proc. of the Intl. Joint Conference on Theory and Practice of Software Development, London, UK, vol. 2, pp. 99–119. Springer, Heidelberg (1991)
4. Brinksma, E.: A theory for the derivation of tests. In: Proc. 8th Int. Conf. Protocol Specification, Testing, and Verification (PSTV VIII), pp. 63–74 (1988)
5. Briones, L., Brinksma, E.: A test generation framework for quiescent real-time systems. In: Grabowski, J., Nielsen, B. (eds.) FATES 2004. LNCS, vol. 3395, pp. 64–78. Springer, Heidelberg (2005)
6. Jéron, T., Jard, C.: TGV: theory, principles and algorithms. International Journal on Software Tools for Technology Transfer 7(4), 297–315 (2005)
7. Fiadeiro, J.L.: Categories for Software Engineering. Springer, Heidelberg (2004)
8. Frantzen, L., Tretmans, J., Willemse, T.A.C.: A Symbolic Framework for Model-Based Testing. In: Havelund, K., Núñez, M., Roşu, G., Wolff, B. (eds.) FATES 2006 and RV 2006. LNCS, vol. 4262, pp. 40–54. Springer, Heidelberg (2006)

9. Gaston, C., Le Gall, P., Rapin, N., Touil, A.: Symbolic execution techniques for test purpose definition. In: Uyar, M.Ü., Duale, A.Y., Fecko, M.A. (eds.) TestCom 2006. LNCS, vol. 3964, pp. 1–18. Springer, Heidelberg (2006)
10. Jifeng, H., Hoare, C.A.R.: Unifying theories of programming. In: Orlowska, E., Szalas, A. (eds.) RelMiCS, pp. 97–99 (1998)
11. Hansen, H.H., Costa, D., Rutten, J.J.M.M.: Synthesis of mealy machines using derivatives. Electr. Notes Theor. Comput. Sci. (ENTCS) 164(1), 27–45 (2006)
12. Hardebolle, C., Boulanger, F.: Exploring multi-paradigm modeling techniques. SIMULATION: Transactions of the Society for Modeling and Simulation International 85(11/12), 688–708 (2009)
13. Heerink, A.W., Tretmans, G.J.: Refusal testing for classes of transition systems with inputs and outputs. In: Mizuno, T., Shiratori, N., Higashino, T., Togashi, A. (eds.) Proceedings of the IFIP TC6 WG6.1 Joint Intl. Conf. on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE X) and Protocol Specification, Testing and Verification (PSTV XVII). IFIP Conference Proceedings, vol. 107, pp. 23–38. Chapman & Hall, Boca Raton (1997)
14. Kanso, B., Aiguier, M., Boulanger, F., Touil, A.: Testing of abstract components. Internal Report 2010-05-28-DI-FBO, Supélec (2010), `http://wwwdi.supelec.fr/`
15. Langerak, R.: A testing theory for LOTOS using deadlock detection. In: Brinksma, E., Scollo, G., Vissers, C.A. (eds.) Protocol Specification, Testing and Verification (PSTV), pp. 87–98. North-Holland, Amsterdam (1989)
16. Lee, D., Yannakakis, M.: Principles and methods of testing finite state machines—a survey. Proceedings of the IEEE 84(8) (August 1996)
17. Mac Lane, S.: Categories for the Working Mathematician. Graduate Texts in Mathematics, vol. 5. Springer, Heidelberg (1971)
18. Mealy, G.H.: A method for synthesizing sequentiel circuits. Bell Systems Techn. Jour. (1955)
19. Meng, S., Barbosa, L.S.: Components as coalgebras: the refinement dimension. Theor. Comput. Sci. (TCS) 351(2), 276–294 (2006)
20. Milner, R.: Communication and concurrency. Prentice-Hall, Inc., Upper Saddle River (1989)
21. Moggi, E.: Notions of computation and monads. Information and Computation 93, 55–92 (1991)
22. De Nicola, R., Hennessy, M.C.B.: Testing equivalences for processes. Theoretical Computer Science (TCS) 34(1-2), 83–133 (1984)
23. Phillips, I.: Refusal testing. Theor. Comput. Sci. 50(3), 241–284 (1987)
24. Sontag, E.D.: Mathematical control theory: deterministic finite dimensional systems, 2nd edn. Springer, New York (1998)
25. Tretmans, J.: A formal approach to conformance testing. In: Proceedings of the IFIP TC6/WG6.1 Sixth International Workshop on Protocol Test systems VI, pp. 257–276. North-Holland Publishing Co., Amsterdam (1994)
26. Tretmans, J.: Test generation with inputs, outputs and repetitive quiescence. Software - Concepts and Tools 17(3), 103–120 (1996)
27. Weilghofer, M., Aichernig, B.: Unifying input output conformance. In: Butterfield, A. (ed.) UTP 2008. LNCS, vol. 5713, pp. 181–201. Springer, Heidelberg (2010)