A formal abstract framework for modeling and testing complex software systems[☆]

Marc Aiguier[a], Frédéric Boulanger[b], Bilal Kanso[a,b],

[a] *École Centrale Paris*
Laboratoire de Mathématiques Appliquées aux Systèmes (MAS)
Grande Voie des Vignes F-92295 Châtenay-Malabry (France)
[b] *SUPELEC Systems Sciences (E3S) - Computer Science Department*
3 rue Joliot-Curie
F-91192 Gif-sur-Yvette cedex (France)

Abstract

The contribution of this paper is twofold: first, it defines a unified framework for modeling abstract components, as well as a formalization of integration rules to combine their behaviour. This is based on a coalgebraic definition of components, which is a categorical representation allowing the unification of a large family of formalisms for specifying state-based systems. Second, it studies compositional conformance testing i.e. checking whether an implementation made of correct interacting components combined with integration operators conforms to its specification.

Keywords: Component based system, Integration operators, Trace semantics, Transfer function, Compositional testing, Conformance testing, Coalgebra, Monad.

Introduction

A powerful approach to developing complex[1] software systems is to describe them in a recursive way as interconnections of subsystems. These subsystems are then integrated through integration operators (so-called architectural connectors) that are powerful tools to describe systems in terms of components and their interactions. Each subsystem can be then either a complex system itself or simple and elementary enough to be handled entirely. However, we observe that components may be designed using different formalisms, most of them being state-based ones. To deal with these heterogeneous components from a global point view, we propose in this paper to model them as concrete coalgebras for an endofunctor on the category of sets following *Barbosa*'s component definition [3, 4]. The interest of such models is twofold: first, defining a component as a coalgebra over the endofunctor $H = T(\mathsf{Out} \times \_)^{\mathsf{In}}$ where T is a monad[2], and In and Out are two sets of elements which denote respectively inputs and outputs of the component, will allow us to abstract away computation situations such as determinism or non-determinism. Indeed, monads have been introduced in [5] to consider in a generic way a wide range of computation structures such as partiality, non-determinism, etc. Hence, *Barbosa*'s definition of components allows us to define components independently of any

[☆] This work is a revised and extended version of [1]. This extension consists in systematically adding proofs of each theorem and proposition given in this paper as well as adding examples to illustrate all the notions introduced in this paper. Moreover, new results about compositional testing are introduced.

Email addresses: `marc.aiguier@ecp.fr` (Marc Aiguier), `frederic.boulanger@supelec.fr` (Frédéric Boulanger), `bilal.kanso@ecp.fr,bilal.kanso@supelec.fr` (Bilal Kanso)

[1] Complex systems are commonly characterized by a holistic behaviour. That means their behaviours cannot be resulted from the combination of isolated behaviours of some of their components, but have to be obtained as whole. This holistic behaviour is classically expressed by the emergence of global properties which are very difficult, even sometimes impossible, to be anticipated just from a complete knowledge of component behaviours [2]. In this paper, the term complex is contrarily used to express component-based systems. We only address here the complexity in terms of heterogeneity of state-based formalisms. Hence, we intend to say by complex systems, the systems described in a recursive way as a set of state-based components, organized and integrated together using integration operators.

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

computation structure. Moreover, this definition will allow us to unify in a same framework a large family of state-based formalisms such as Mealy automata [6, 7], Labeled Transition Systems [8], Input-Output Symbolic Transition Systems [9, 10, 11], etc. Second, following *Rutten*'s works [12], defining component behaviours as extensions 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 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 is discrete time.

Hence, by extending *Rutten*'s results in [12] on the existence of a final coalgebra in the category of coalgebras over $H = T(\mathsf{Out} \times \_)^{\mathsf{In}}$, we will show how to define causal functions which underly any system, or equivalently, that the system under consideration implements. This constitutes our first contribution in this paper. Another contribution in this paper is the definition of two basic integration operators, *product* and *feedback*, which are used to build larger systems by composition of subsystems. Indeed, we defend the idea that most standard integration operators can be obtained by composition of product and feedback, and we will show this for them. This will lead us to define inductively more complex integration operators, the semantics of which will be partial functors over categories of components. Hence, a system will be built by a recursive hierarchical process through these integration operators from elementary systems or basic components.

Finally, as a last contribution in this paper, we propose to define a conformance testing theory for components. From the generality of the formalism developed in this paper, the testing theory developed here will be *de facto* applicable to all state-based formalisms, instances of our framework such as these presented in Section 2.2. There are several conformance testing theory in the literature [9, 10, 13, 14, 15, 16] that differ by the considered conformance relation and algorithms used to generate test cases. Although most of these theories could be adapted to our formalism, we propose here to extend the approach defined in [10] in the context of $IOSTS$ formalism. The advantage of the testing theory proposed in [10] is that it is based on the conformance relation *ioco* that received much attention by the community of formal testing because it has shown its suitability for conformance testing and automatic test derivation. Furthermore, test generation algorithms proposed in [10] are simple in their implementation and efficient in their execution. Hence, 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 will generate test cases from test purposes. 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 stimulation sent by the test case to the SUT. This testing process leads to a verdict.

This conformance testing theory is a step toward the testing of complex software systems made of interacting components. In the present paper, we further propose to define a compositional testing theory that aims to check whether the correctness of a whole system $C = op(C_1, \ldots, C_n)$ is established using the correctness of each component C_i, where op is any integration operator. Hence, the problem of compositional testing can be seen as follows: given implementation models $I_1, \ldots, I_n$, their specifications $C_1, \ldots, C_n$, an integration operator op and a conformance relation rel such as for every $i, 1 \leq i \leq n$, I_i has been tested to be rel-correct according to its specification C_i, can we conclude that their composition $op(I_1, \ldots, I_n)$ is also rel-correct with respect to the integrated specification $op(C_1, \ldots, C_n)$? A positive answer to this question cannot be obtained without any assumption on both specifications and implementations. We will show in this paper that under some conditions, the conformance relation is preserved along any integration operator. These last results extend *Tretman*'s results exposed in [17] to our framework.

The paper is structured as follows: Section 1 recalls the basic notions of coalgebras and monads that will be useful in this paper. Section 2 recalls *Barbosa*'s definition of components and defines a trace model from causal transfer functions over it. The formalization of components as coalgebras allows us to extend some standard results connected to the definition of a final component in Section 3. Section 4 presents the basic integration operators: cartesian product and both relaxed and synchronous feedback as well as how to combine them to build more complex integration operators. In Section 5, we define the notion of system that will be the result of the composition of basic components using complex integration operators. Section 6 presents our generic conformance testing theory for components. Section 7 gives the algorithm for generating test cases as a set of inference rules. Section 8 studies the preservation of the conformance relation by the proposed basic integration operators.

1. Preliminaries

This paper relies on many terms and notations from the categorical theory of coalgebras and monads. The notions introduced here make use of basic notions of category theory (category, functors, natural transformations, etc.) We

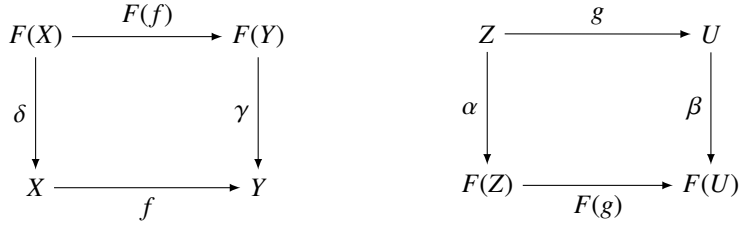
do not present these notions in this preliminaries, but interested readers may refer to textbooks such as [18, 19, 20]. Similarly, readers wanting to go into details in coalgebras and monads may refer to [20, 21]

1.1. 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 mapping $\alpha : F(X) \to X$. In this categorical definition, F gives the signature of the algebra. For instance, with $\mathbf{1}$ denoting class of isomorphism of the singleton set $\{\star\}$, by considering the functor $F = \mathbf{1} + \_$ which maps $X \mapsto 1 + X$, 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 mapping $\alpha : X \to F(X)$. 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 object. For instance, a Mealy machine with state set S and input and output alphabets In and Out, can be described as an F-coalgebra $(S, \langle \mathsf{out}, \mathsf{next} \rangle)$ of the functor $F = (\mathsf{Out} \times \_)^{\mathsf{In}}$. The $\mathsf{out} : S \to \mathsf{Out}^{\mathsf{In}}$ operation is the curried form of the output function $\mathsf{out} : S \times \mathsf{In} \to \mathsf{Out}$ which associates an output to an input when the machine is in a given state. Similarly next is the curried form of the transition function $\mathsf{next} : S \times \mathsf{In} \to S$ which associates a new state to an input when the machine is in a given state.

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 (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 (Ω, δ) 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 an F-algebra (X, γ) defines in a unique way the homomorphism $!_\gamma : \Omega \to X$ from the initial F-algebra (Ω, δ) to this algebra.

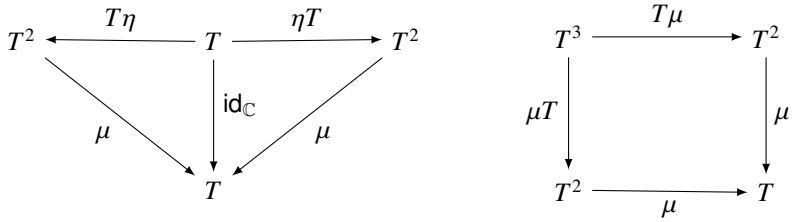
Conversely, if a final coalgebra (Γ, π) exists in $\mathbf{CoAlg}(F)$, it is unique, and its structure map is an isomorphism. The uniqueness of the homomorphism from any object to a final object of a category is the key for defining morphisms by coinduction: giving an F-coalgebra (Y, α) defines in a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

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 [22]. 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.

1.2. Monads

Monads are a powerful abstraction for adding structure to objects and arrows. 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 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 as a structured object $T(\eta_X)(x)$ made of the 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 $T(\mu_X)$).

As an example, 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 for S has to build a set of states from a state. We obviously 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 for S has to flatten a set of state sets 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 $\mu_S : \{s_1 \dots s_i \dots\} \mapsto \cup s_i$. It is easy to check that η and μ are natural transformations, such as for any morphism $f : X \to Y$, $\mathcal{P}(f) : \mathcal{P}(X) \to \mathcal{P}(Y)$ is the morphism which maps each part of X to its image by f.

In computer science, monads have mainly been used to present many computation situations such as partiality, side-effects, exceptions, etc. For instance, partiality can be represented by the monad $T : \mathsf{id} + \mathbf{1}$ over **Set** equipped with both obvious natural transformations η and μ, which for any set S are defined by:

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

Many other examples can be found in [5].

2. Transfer functions and components

2.1. Transfer function

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

Definition 2.1 (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 will be used to describe the observable behaviour of components. They can be seen as dataflow transformers satisfying the causality condition as this is classically done in control theory and physics to modeling dynamic systems [23], that is the output data at index n only depends on input data at indexes $0, \dots, n$.

Definition 2.2 (Transfer function). *Let In and Out be two sets denoting, respectively, the input and output domains. A function $\mathcal{F} : \mathsf{In}^\omega \longrightarrow \mathsf{Out}^\omega$ is a **transfer function** if, and only if it is causal, that is:*

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

Example 2.1. *The function $\mathcal{F} : \{0,1\}^\omega \longrightarrow \{0,1\}^\omega$ defined for every $\sigma \in \{0,1\}^\omega$ and every $k \in \omega$ by*

$$\mathcal{F}(\sigma)(k) = \Big(\sum_{i=0}^{k} \sigma(i) \Big) \bmod 2$$

is the transfer function that takes a sequence of bits $\sigma \in \{0,1\}^\omega$ and checks at each step k whether it has received an odd number of ones. It then returns 0 if the one's number is even, and 1 otherwise. In Example 3.1, we will define the component that implements it.

2.2. Components

Definition 2.3 (Components). *Let* In *and* Out *be two sets denoting, respectively, the input and output domains. Let* T *be a monad. A* ***component*** C *is a coalgebra* (S, α) *for the signature* $H = T(\mathsf{Out} \times \_)^{\mathsf{In}} : \mathbf{Set} \longrightarrow \mathbf{Set}$ *with a distinguished element* $init \in S$ *denoting the initial state of the component* C.
When the initial state $init$ *is removed,* C *is called* ***a pre-component***.

Example 2.2. *We illustrate the notions previously mentioned with the simple example of a coffee machine* $\mathcal{M}$ *modeled by the transition diagram shown on Figure 1. The behaviour 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,* $\mathcal{M}$ *either serves a coffee to the user and goes back 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*[3] $\mathcal{P}_{\text{fin}}(\mathsf{Out} \times \_)^{\mathsf{In}}$. *The state space is* $S = \{\text{STDBY}, \text{READY}, \text{FAILED}\}$ *and* $s_0 = \text{STDBY}$. *The sets of inputs and outputs are* $\mathsf{In} = \{\text{coin}, \text{coffee}, \text{repair}\}$ *and* $\mathsf{Out} = \{\mathsf{abs}, \text{served}, \text{refund}\}$. *Finally, the transition function:*

$$\alpha : S \longrightarrow \mathcal{P}_{\text{fin}}\Big(\{\mathsf{abs}, \text{served}, \text{refund}\} \times S\Big)^{\{\text{coin},\text{coffee},\text{repair}\}}$$

is defined as follows:

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

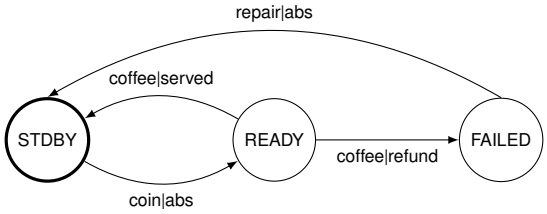


Figure 1: Coffee machine

Using Definition 2.3 for components, we can unify in a single 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 Id, the resulting component corresponds to a Mealy machine. Choosing $\mathsf{Out} = \{\mathsf{abs}\}$ and $\mathsf{In} = Act$, a set of symbols standing for actions names, and with the powerset monad $\mathcal{P}$ for T, the resulting component corresponds to a Labeled Transition System. Finally, taking as the powerset monad $\mathcal{P}$ for T, $\mathsf{In} = (\{?\} \times \Sigma^{?}) \cup \{\mathsf{abs}_?\}$, $\mathsf{Out} = (\{!\} \times \Sigma^{!}) \cup \{\mathsf{abs}_!\}$ and imposing[4] the supplementary property on the transition function $\alpha : S \longrightarrow \mathcal{P}(\mathsf{Out} \times S)^{\mathsf{In}}$:

$$\forall i \in \mathsf{In}, \forall s \in S, (o, s') \in \alpha(s)(i) \Longrightarrow \text{ either } i = \mathsf{abs}_? \text{ or } o = \mathsf{abs}_!$$

leads to IOLTS.

Definition 2.4 (Category of components). *Let* C *and* C' *be two components over* $H = T(\mathsf{Out} \times \_)^{\mathsf{In}}$. *A* ***component morphism*** $h : C \to C'$ *is a coalgebra homomorphism* $h : (S, \alpha) \to (S', \alpha')$ *such that* $h(init) = init'$.
We note $\mathbf{Comp}(H)$ *(resp.* $\mathbf{PComp}(H)$*) the* ***category of components*** *(resp. pre-components) over* H.

The category of pre-components $\mathbf{PComp}(H)$ will be useful for us to ensure, in the next section, the existence of a final model.

[3] $\mathcal{P}_{\text{fin}}(X) = \{U \subseteq X | U \text{ is finite}\}$ is the finite powerset of X.
[4] $\mathsf{abs}_?, \mathsf{abs}_! \notin \Sigma^{!} \cup \Sigma^{!}$ are particular fresh input and output actions denoting the lack of input ($\mathsf{abs}_?$) and output ($\mathsf{abs}_!$).

2.3. Component Traces

To associate behaviours to components by transfer functions, we have to require the existence of two natural transformations $\eta' : T \Longrightarrow \mathcal{P}$ and $\eta'^{-1} : \mathcal{P} \Longrightarrow T$ such that $\eta'^{-1} \circ \eta' = \mathsf{id}_T$ where $\mathcal{P}$ is the powerset functor. Indeed, from a component (S, α), we need to "compute" for a sequence $x \in \mathsf{In}^\omega$ all the outputs o after going through 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 $\alpha(s_{j-1})(x(j-1))$ because nothing ensures that elements in $\alpha(s_{j-1})(x(j-1))$ are (output, state) couples. Indeed, the monad T may yield a set with a structure which differs from $\mathsf{Out} \times S$. The mapping $\eta'_{\mathsf{Out}\times S}$ maps back to this structure. $\eta'^{-1}_{\mathsf{Out}\times S}$ is useful for going back to T when defining final models.

Most monads used to represent computation situations satisfy the above condition. For instance, for the monad $T : \mathcal{P}$, both η'_S and η'^{-1}_S are the identity on sets. For the functor $T : \mathsf{id} + \mathbf{1}$, η'_S associates the singleton $\{s\}$ to any $s \in S$ and the empty set to $\perp$, and η'^{-1}_S associates the state s to the singleton $\{s\}$ and $\perp$ to any other subset of S which is not a singleton. Let us observe that given a monad T, the couple (η', η'^{-1}) when it exists, is not necessarily unique. Indeed, for the monad $T = \mathsf{id}$, η'_S can still be defined as $s \mapsto \{s\}$. However, η'^{-1}_S is not unique. Indeed, any mapping η'^{-1}_S that associates the singleton $\{s\}$ to s, and every subset of S which is not a singleton to a given $s' \in S$, satisfies $\eta'^{-1}_S \circ \eta'_S = \mathsf{id}_S$. Hence, in the following, for any signature $T(\mathsf{Out} \times \_)^{\mathsf{In}}$, we will assume that a couple (η', η'^{-1}) such that $\eta'^{-1} \circ \eta' = \mathsf{id}$ is given.

In the following, we note $\eta'_{\mathsf{Out}\times S}(\alpha(s)(i))_{|_1}$ (*resp.* $\eta'_{\mathsf{Out}\times S}(\alpha(s)(i))_{|_2}$) the set composed of all first arguments (*resp.* second arguments) of couples in $\alpha(s)(i)$.

Let us now associate behaviours to components by their transfer functions. Let us consider a state $s \in S$ of such a component $C = (S, \alpha)$ over $T(\mathsf{Out} \times \_)^{\mathsf{In}}$. Applying α to s after receiving an input $i_1 \in \mathsf{In}$ yields a set $\eta'_{\mathsf{Out}\times S}(\alpha(s)(i_1))$ of couples (output|successor state). Similarly, after receiving a new input $i_2 \in \mathsf{In}$, we can repeat this step for each state $s' \in \eta'_{\mathsf{Out}\times S}(\alpha(s)(i_1))_{|_2}$ and form another set of couples (output|successor state). Thus, we get for each infinite sequence of inputs $\langle i_1, i_2, \ldots\rangle \in \mathsf{In}^\omega$, a set of infinite sequences of outputs $\langle o_1, o_2, \ldots\rangle \in \mathsf{Out}^\omega$. All we can possibly observe about a state $s \in S$ is obtained in this way. More formally, this leads to:

Definition 2.5 (Component behaviour). *Let C be a component over $T(\mathsf{Out} \times \_)^{\mathsf{In}}$. The **behaviour** of a state s of C, denoted by $\mathsf{beh}_C(\mathbf{s})$ is the set of transfer functions $\mathcal{F} : \mathsf{In}^\omega \longrightarrow \mathsf{Out}^\omega$ that associate to every $x \in \mathsf{In}^\omega$ any dataflow $y \in \mathsf{Out}^\omega$ such that there exists an infinite sequence of couples $(o_1, s_1), \ldots, (o_k, s_k), \ldots \in \mathsf{Out} \times S$ satisfying:*

$$\forall j \geq 1, (o_j, s_j) \in \eta'_{\mathsf{Out}\times S}(\alpha(s_{j-1})(x(j-1)))$$

with $s_0 = s$, and for every $k < \omega$, $y(k) = o_{k+1}$.
*Hence, C's **behaviour** is the set $\mathsf{beh}_C(init)$.*

Example 2.3. *The behaviour $beh_{\mathcal{M}}(s_0)$ of the coffee machine $\mathcal{M}$ presented in Example 2.2 is defined by all the functions $\mathcal{F}_\sigma : \{\mathsf{coin}, \mathsf{coffee}, \mathsf{repair}\}^\omega \longrightarrow \{\mathsf{abs}, \mathsf{served}, \mathsf{refund}\}^\omega$ where $\sigma = n_1.n'_1.n_2.n'_2 \ldots n_i.n'_i \ldots \in \mathbb{N}^\omega$ defined by*

$\mathcal{F}_\sigma : (\mathsf{coin.coffee})^{n_1}.(\mathsf{coin.coffee.repair})^{n'_1} \ldots (\mathsf{coin.coffee})^{n_i}.(\mathsf{coin.coffee.repair})^{n'_i} \ldots$
$\mapsto$
$(\mathsf{abs.served})^{n_1}.(\mathsf{abs.refund.abs})^{n'_1} \ldots (\mathsf{abs.served})^{n_i}.(\mathsf{abs.refund.abs})^{n'_i} \ldots$

where

$(\mathsf{coin.coffee})^0 = (\mathsf{coin.coffee.repair})^0 = (\mathsf{abs.served})^0 = (\mathsf{abs.refund.abs})^0 = \epsilon$ *(the empty word).*

Hence, the transfer function that would remain in the loop between the states STDBY *and* READY *would be able to be defined by any function $\mathcal{F}_\sigma$ with $\sigma = n_1.0.n_2.0 \ldots n_i.0 \ldots$*

Thus, the behaviour mapping beh associates to every state $s \in S$ a set of causal functions $\mathsf{beh}_C(s)$. We will show in Section 3 that the set of all causal function sets can be equipped with the structure of a pre-component via the notion of derivative function. Moreover, this pre-component will be shown to be final in the category $\mathbf{PComp}(H)$.

3. Final model of a component

We show here that under some conditions on the cardinality of the set yielded by the mapping beh_C for every component $C = (S, \alpha) \in \mathbf{PComp}(H)$, the coalgebra of all causal function sets is final.

3.1. Final model

If we suppose that for every pre-component $C = (S, \alpha)$ over a signature $H = T(\mathsf{Out} \times \_)^{\mathsf{In}}$, and for every $s \in S$ the cardinality of $\mathsf{beh}_C(s)$ is less than a cardinal κ, then we can define a coalgebra (Γ, π) over H and show that it is final in $\mathbf{PComp}(H)$. But before, let us introduce some notions that will be useful for this purpose.

Definition 3.1 (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 denote by $a.x$ 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 3.2 (Derivative function). *Let $\mathcal{F} : \mathsf{In}^\omega \longrightarrow \mathsf{Out}^\omega$ be a transfer function. For every input $i \in \mathsf{In}$, we define the* **derivative function** *$\mathcal{F}_i : \mathsf{In}^\omega \longrightarrow \mathsf{Out}^\omega$ for every $x \in \mathsf{In}^\omega$ by $\mathcal{F}_i(x) = \mathcal{F}(i.x)'$.*

Hence, given a cardinality κ, let us now define the coalgebra (Γ, π) as follows: [5]

- $\Gamma = \mathcal{P}_{\leq\kappa}(\{\mathcal{F} : \mathsf{In}^\omega \longrightarrow \mathsf{Out}^\omega \mid \mathcal{F} \text{ is causal}\})$
- for every $\mathbb{F} \in \Gamma$ and for every $i \in \mathsf{In}$, $\pi(\mathbb{F})(i) = \eta'^{-1}_{\mathsf{Out}\times\Gamma}(\Pi)$ where:

$$\Pi = \Big\{(o, \mathbb{F}'_o) \quad \Big| \quad o \in \bigcup_{\mathcal{F}\in\mathbb{F}} (\mathcal{F}(i.x)(0)) \text{ and},\ \mathbb{F}'_o = \{\mathcal{F}(i.x)' \mid \mathcal{F}(i.x)(0) = o \text{ and } \mathcal{F} \in \mathbb{F}\},\ \text{for } x \in \mathsf{In}^\omega \text{ chosen arbitrarily}\Big\}$$

Let us note here that using $\mathbb{F}'_o = \{\mathcal{F}(i.x)' \mid \mathcal{F}(i.x)(0) = o \text{ and } \mathcal{F} \in \mathbb{F}\}$ instead of $\mathbb{F}' = \{\mathcal{F}(i.x)' \mid \mathcal{F} \in \mathbb{F}\}$ in the definition of (Γ, π) allows us to keep the computational effects carried by the monad T. This is done by linking the output o to the derivative function set $\mathbb{F}'$ *i.e.* the derivative function set is not only linked to the input i but also to the output associated to i. This construction of the set $\mathbb{F}'$ is useful to prove that (Γ, π) is final in $\mathbf{PComp}(H)$.

Theorem 3.1. *Let $H = T(\mathsf{Out} \times \_)^{\mathsf{In}}$ be a signature such that for every pre-component $C = (S, \alpha)$ over H, and for every $s \in S$, $|\mathsf{beh}_C(s)| \leq \kappa$. Then, the coalgebra (Γ, π) is final in $\mathbf{PComp}(H)$.*

Proof . *Let (Γ, π) be as stated, and let $C = (S, \alpha) \in \mathbf{PComp}(H)$ be an arbitrary component. We have to show that there exists a unique homomorphism of components $S \to \Gamma$. For this, let us take the behaviour mapping $\mathsf{beh}_C : S \to \Gamma$ (see Definition 2.5) which for every $s \in S$ associates a finite set of transfer functions $\mathbb{F} = \{\mathcal{F} : \mathsf{In}^\omega \longrightarrow \mathsf{Out}^\omega \mid \mathcal{F} \text{ is causal}\} \in \Gamma$. We have to prove that it is the unique homomorphism which makes the following diagram commute.*

$$\begin{array}{ccc} S \times \mathsf{In} & \xrightarrow{\mathsf{beh}_C \times \mathsf{id}_{\mathsf{In}}} & \Gamma \times \mathsf{In} \\ \Big\downarrow{\scriptstyle \alpha} & & \Big\downarrow{\scriptstyle \pi} \\ T(\mathsf{Out} \times S) & \xrightarrow[T(\mathsf{id}_{\mathsf{Out}} \times \mathsf{beh}_C)]{} & T(\mathsf{Out} \times \Gamma) \end{array}$$

[5] $\mathcal{P}_{\leq\kappa}(X) = \{U \mid U \subseteq X \text{ and } |U| \leq \kappa\}$

We first prove that the diagram commutes.
First of all, it is easy to see that the three following properties are satisfied:

$$\forall f : S \longrightarrow S', \forall X \in T(S) : \; T(f)(X) = \eta'^{-1}_{S'} \circ \mathcal{P}(f) \circ \eta'_S(X) \tag{1}$$

$$\forall s \in S, i \in \mathsf{In} \textit{ and } \forall (o, s') \in \eta'_{\mathsf{Out}\times S}(\alpha(s)(i)) \quad \mathsf{beh}_C(s') = \mathsf{beh}_C(s)'_o = \{\mathcal{F}(i.x)' \mid \mathcal{F}(i.x)(0) = o \text{ and } \mathcal{F} \in \mathsf{beh}_C(s)\} \tag{2}$$

$$\forall i \in \mathsf{In}, s \in S \textit{ and } \mathsf{beh}_C(s) \in \Gamma, \quad \eta'_{\mathsf{Out}\times S}(\alpha(s)(i))_{|_1} = \{\mathcal{F}(i.x)(0) \mid \mathcal{F} \in \mathsf{beh}_C(s)\} \tag{3}$$

Hence, let $s \in S$, $i \in \mathsf{In}$ and $x \in \mathsf{In}^\omega$ be arbitrary. We have to prove that:

$$(T(\mathsf{id}_{\mathsf{Out}} \times \mathsf{beh}_C) \circ \alpha)(s)(i) = (\pi \circ (\mathsf{beh}_C \times \mathsf{id}_{\mathsf{In}}))(s)(i)$$

$$\begin{aligned}
(T(id_{\mathsf{Out}} \times \mathsf{beh}_C) \circ \alpha)(s)(i) &= T(\mathsf{id}_{\mathsf{Out}} \times \mathsf{beh}_C)(\alpha(s)(i)) \\
&= \eta'^{-1}_{\mathsf{Out}\times\Gamma}(\mathcal{P}(\mathsf{id}_{\mathsf{Out}} \times \mathsf{beh}_C)(\eta'_{\mathsf{Out}\times S}(\alpha(s)(i)))) \qquad \text{Property 1} \\
&= \eta'^{-1}_{\mathsf{Out}\times\Gamma}(\{(o, \mathsf{beh}_C(s')) \mid (o, s') \in \eta'_{\mathsf{Out}\times S}(\alpha(s)(i))\}) \quad \text{Property 2} \\
&= \eta'^{-1}_{\mathsf{Out}\times\Gamma}\Big\{(o, \mathsf{beh}_C(s)'_o) \mid o \in \eta'_{\mathsf{Out}\times S}(\alpha(s)(i))_{|_1} \text{and}, \\
&\qquad \mathsf{beh}_C(s)'_o = \{\mathcal{F}(i.x)' \mid \mathcal{F}(i.x)(0) = o \text{ and } \mathcal{F} \in \mathsf{beh}(s)\}\Big\} \quad \text{Property 3} \\
&= \eta'^{-1}_{\mathsf{Out}\times\Gamma}\Big\{(o, \mathsf{beh}_C(s)'_o) \mid o \in \mathcal{F} \in \mathsf{beh}_C(s)(\mathcal{F}(i.x)(0)) \text{ and}, \\
&\qquad \mathsf{beh}_C(s)'_o = \{\mathcal{F}(i.x)' \mid \mathcal{F}(i.x)(0) = o \text{ and } \mathcal{F} \in \mathsf{beh}_C(s)\}\Big\} \text{ Def. of } \pi \\
&= \pi((\mathsf{beh}_C(s), i)) \\
&= \pi(\mathsf{beh}_C \times \mathsf{id}_{\mathsf{In}})(s, i) \\
&= (\pi \circ (\mathsf{beh}_C \times \mathsf{id}_{\mathsf{In}}))(s, i)
\end{aligned}$$

Next we have to prove uniqueness. In order to prove this last point, we need to prove the following lemma:

Lemma 1. *For every component homomorphism $f : S \to \Gamma$, for every $x \in \mathsf{In}^\omega$ and for every $s \in S$ we have:*

$$(f(s)(x))' = \{f(s')(x') \mid s' \in \eta'_{\mathsf{Out}\times S}(\alpha(s)(x(0)))_{|_2}\}$$

where x' is the derivative of x.

Proof .

$$
\begin{aligned}
(f(s)(x))' &= \Big\{(o_1, o_2, \ldots, o_k, \ldots)' \mid \exists s_0, s_1, \ldots, s_k, \ldots \in S \\
&\quad \text{such that } s = s_0,\ \langle o_1, s_1\rangle \in \eta'_{\mathsf{Out}\times S}(\alpha(s_0)(x(0))) \\
&\quad \text{and } \forall 2 \le j \le k-1, (o_j, s_j) \in \eta'_{\mathsf{Out}\times S}((s_{j-1})(x(j-1))), \\
&\quad \text{and } o_k \in \eta'_{\mathsf{Out}\times S}(\alpha((s_k)(x(k)))_{|_1})\Big\} \\
&= \Big\{(o_2, \ldots, o_k, \ldots) \mid \exists s_1, \ldots, s_k, \ldots \in S \\
&\quad \text{such that } s_1 \in \eta'_{\mathsf{Out}\times S}(\alpha(s_0)(x(0)))_{|_2} \\
&\quad \text{and } \forall 2 \le j \le k-1, s_j \in \eta'_{\mathsf{Out}\times S}(\alpha((s_{j-1})(x(j-1))))_{|_2}, \\
&\quad \text{and } o_k \in \eta'_{\mathsf{Out}\times S}(\alpha(s_k)(x(k)))_{|_1}\Big\} \\
&= \{f(s_1)(x') \mid s_1 \in \eta'_{\mathsf{Out}\times S}(\alpha(s_0)(x(0))_{|_2}\} \\
&= \{f(s')(x') \mid s' \in \eta'_{\mathsf{Out}\times S}(\alpha(s)(x(0))_{|_2}\}
\end{aligned}
$$

End

Now, let us assume that $g : S \to \Gamma$ is also a homomorphism of components. Let us show that the relation $R \subseteq \mathcal{P}_\kappa(\mathsf{Out}^\omega) \times \mathcal{P}_\kappa(\mathsf{Out}^\omega)$ defined as:

$$R = \{\langle g(s)(x), \mathsf{beh}(s)(x)\rangle \mid s \in S, x \in \mathsf{In}^\omega,\ g(s)(x) = \mathsf{beh}(s)(x)\}$$

is a bisimulation.
It can be shown by coinduction on $x \in \mathsf{In}^\omega$, that for all $s \in S$ we have:

$$g(s)(x) = \mathsf{beh}(s)(x)$$

The initial set outputs of $g(s)(x)$ and $\mathsf{beh}(s)(x)$ agree, since at the initial input $x(0)$ of x, we have:

$$g(s)(x)(0) = \eta'_{\mathsf{Out}\times S}(\alpha(s)(x(0))_{|_1} = \mathsf{beh}(s)(x)(0)$$

$$(g(s)(x))' = (g(s)(x(0).x'))' = \{g(s')(x') \mid s' \in \eta'_{\mathsf{Out}\times S}(\alpha(s)(x(0)))_{|_2}\} \quad \textit{Lemma } 2$$

$$(\mathsf{beh}(s)(x))' = (\mathsf{beh}(s)(x(0).x'))' = \{\mathsf{beh}(s')(x') \mid s' \in \eta'_{\mathsf{Out}\times S}(\alpha(s)(x(0)))_{|_2}\} \quad \textit{Lemma } 2$$

Hence the function derivatives sets are also R-related, and we conclude that R is a bisimulation.
End

3.2. Minimal component

A final model of the functor $F = T(\mathsf{Out} \times \_)^{\mathsf{In}}$ provides an abstract model of all possible behaviours of its F-coalgebras. Hence, in practice, it cannot be handled as a whole, but we can construct the minimal part of it (minimality refers to the cardinality of the state set) for every state $s \in S$ of a F-coalgebra $C = (S, \alpha)$. This is done by computing the smallest subcoalgebra in (Γ, π) containing $\mathsf{beh}_C(s)$. More generally, given a subset $\mathbb{F} \in \Gamma$ of causal functions, we can compute the smallest subcoalgebra in (Γ, π), noted $\langle\mathbb{F}\rangle$, containing $\mathbb{F}$. This coalgebra is called the *coalgebra generated by* $\mathbb{F}$ in (Γ, π).

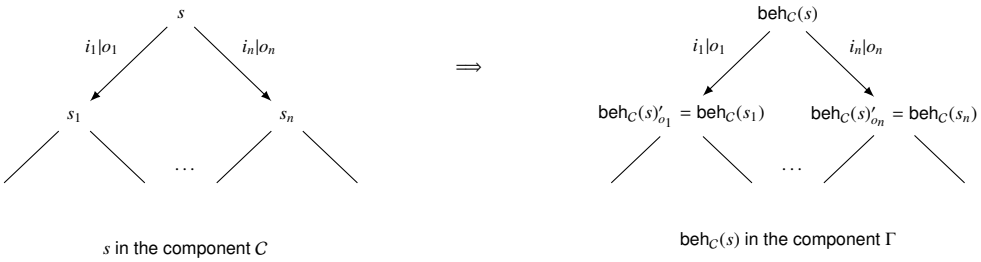
Definition 3.3 (Component generated by $\mathbb{F}$). *Let (Γ, π) be the final model over the signature $H = T(\mathsf{Out} \times \_)^{\mathsf{In}}$. Let $\mathbb{F} \in \Gamma$. The **component** $\langle\mathbb{F}\rangle$ **generated by** $\mathbb{F}$ in (Γ, π) is the component $(\langle\mathbb{F}\rangle, \mathbb{F}, \alpha_{\langle\mathbb{F}\rangle})$ defined as follows:*

- $\mathbb{F}$ *is the initial state,*
- $\langle\mathbb{F}\rangle$ *is the set of transfer functions sets inductively defined as follows:*
  - $\langle\mathbb{F}\rangle^0 = \{\mathbb{F}\}$
  - $\langle\mathbb{F}\rangle^j = \Big\{\mathbb{G}' \mid \exists\mathbb{G} \in \langle\mathbb{F}\rangle^{j-1}, \exists i \in \mathsf{In}, \exists o \in \mathsf{Out}, o \in \bigcup_{\mathcal{F}\in\mathbb{G}} \mathcal{F}(i.x)(0)$
    $\text{and } \mathbb{G}' = \{\mathcal{F}(i.x)' \mid \mathcal{F}(i.x)(0) = o \text{ and } \mathcal{F} \in \mathbb{G}\},$
    $\text{for } x \in \mathsf{In}^\omega \text{ chosen arbitrarily}\Big\}$

  Hence, $\langle\mathbb{F}\rangle = \bigcup_{j<\omega} \langle\mathbb{F}\rangle^j$
- $\alpha_{\langle\mathbb{F}\rangle} : \langle\mathbb{F}\rangle \times \mathsf{In} \to T(\mathsf{Out} \times \langle\mathbb{F}\rangle)$ *is the mapping which for every* $\mathbb{G} \in \langle\mathbb{F}\rangle$*, and for every input* $i \in \mathsf{In}$ *associates* $\eta^{-1}_{\mathsf{Out}\times\langle\mathbb{F}\rangle}(\Pi')$ *where* Π' *is the set:*

$$\Pi' = \Big\{(o, \mathbb{G}'_o) \quad | \quad o \in \bigcup_{\mathcal{F}\in\mathbb{G}} (\mathcal{F}(i.x)(0)) \text{ and},\ \mathbb{G}'_o = \{\mathcal{F}(i.x)' \mid \mathcal{F}(i.x)(0) = o \text{ and } \mathcal{F} \in \mathbb{G}\},\ \text{for } x \in \mathsf{In}^\omega \text{ chosen arbitrarily}\Big\}$$

It is easy to notice that both components $C = (S, init, \alpha)$ and $\langle\mathsf{beh}_C(init)\rangle$ share the same trace semantics i.e. $Trace(C) = Trace(\langle\mathsf{beh}_C(init)\rangle) = \mathsf{beh}_C(init)$. (see Definition 2.5).



Example 3.1. *For a better understanding of the definition of a minimal component, we consider as an example the binary Mealy machine* $\mathcal{M}$ *modeled by the transition diagram shown on Figure 2. This machine* $\mathcal{M}$ *is considered as a component* $\mathcal{M} = (\{s_0, s_1, s_2\}, s_0, \alpha)$ *over the signature* $(\{0,1\} \times \_)^{\{0,1\}}$ *where the transition function:*

$$\alpha : \{s_0, s_1, s_2\} \longrightarrow (\{0,1\} \times \{s_0, s_1, s_2\})^{\{0,1\}}$$

is defined as follows:

$$\begin{cases} \alpha(s_0)(0) = (0, s_2) \\ \alpha(s_0)(1) = (1, s_1) \end{cases} \quad \begin{cases} \alpha(s_1)(0) = (1, s_1) \\ \alpha(s_1)(1) = (0, s_2) \end{cases} \quad \begin{cases} \alpha(s_2)(0) = (0, s_2) \\ \alpha(s_2)(1) = (1, s_1) \end{cases}$$

It is not difficult to see that applying Definition 2.5 to the initial state s_0 *leads to the minimal set of transfer functions* $\mathsf{beh}_{\mathcal{M}}(s_0) = \{\mathcal{F}_1\}$ *where* $\mathcal{F}_1 : \{0,1\}^\omega \longrightarrow \{0,1\}^\omega$ *is the transfer function of Example 2.1, i.e. the one defined for every* $\sigma \in \{0,1\}^\omega$ *and for every* $k \in \omega$ *by:*

$$\mathcal{F}_1(\sigma(k)) = \Big(\sum_{i=0}^{k} \sigma(i)\Big) \bmod 2$$

Now to compute the minimal component $\langle\mathsf{beh}_{\mathcal{M}}(s_0)\rangle$*, we need to compute all derivative sets of transfer functions starting from* $\mathsf{beh}_{\mathcal{M}}(s_0)$*. With a simple computing, we can conclude that the state of* $\langle\mathsf{beh}_{\mathcal{M}}(s_0)\rangle$ *consists of two states:*

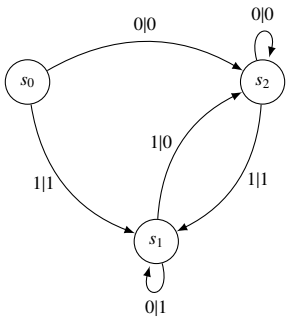


Figure 2: Binary Mealy automaton

$\{\mathcal{F}_1\}$ and $\{\mathcal{F}_2\}$ where $\mathcal{F}_2 : \{0,1\}^\omega \longrightarrow \{0,1\}^\omega$ is the transfer function defined for every $\sigma \in \{0,1\}^\omega$ and for every $k \in \omega$ by:

$$\mathcal{F}_2(\sigma(k)) = 1 - \Big(\sum_{i=0}^{k} \sigma(i)\Big) \bmod 2$$

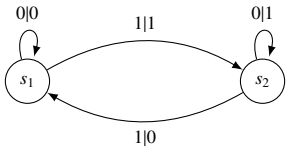
Computing further derivative sets will not yield any new transfer functions sets. Thus, $\langle \mathsf{beh}_{\mathcal{M}}(s_0)\rangle$ is the component $(\{\mathcal{F}_1, \mathcal{F}_2\}, \{\mathcal{F}_1\}, \alpha_{\mathsf{beh}_{\mathcal{M}}(s_0)})$ where:

$$\alpha_{\mathsf{beh}_{\mathcal{M}}(s_0)} : \{\{\mathcal{F}_1\}, \{\mathcal{F}_2\}\} \longrightarrow (\{0,1\} \times \{\{\mathcal{F}_1\}, \{\mathcal{F}_2\}\})^{\{0,1\}}$$

is the transition function defined as follows:

$$\begin{cases} \alpha_{\mathsf{beh}_{\mathcal{M}}(s_0)}(\{\mathcal{F}_1\})(0) = (0, \{\mathcal{F}_1\}) \\ \alpha_{\mathsf{beh}_{\mathcal{M}}(s_0)}(\{\mathcal{F}_1\})(1) = (1, \{\mathcal{F}_2\}) \\ \alpha_{\mathsf{beh}_{\mathcal{M}}(s_0)}(\{\mathcal{F}_2\})(0) = (1, \{\mathcal{F}_2\}) \\ \alpha_{\mathsf{beh}_{\mathcal{M}}(s_0)}(\{\mathcal{F}_2\})(1) = (0, \{\mathcal{F}_1\}) \end{cases}$$

and then can be depicted as:



4. Integration of components

4.1. Basic integration operators

4.1.1. Cartesian product

The cartesian product is a composition where both components are executed simultaneously when triggered by a pair of input values.

Definition 4.1 (Cartesian product $\otimes$). *Let $H_1 = T(\mathsf{Out}_1 \times \_)^{\mathsf{In}_1}$ and $H_2 = T(\mathsf{Out}_2 \times \_)^{\mathsf{In}_2}$ be two signatures. Let $H = T((\mathsf{Out}_1 \times \mathsf{Out}_2) \times \_)^{(\mathsf{In}_1 \times \mathsf{In}_2)}$ be the signature resulting of the product of H_1 and H_2. Let us define the **cartesian integration functor**:*

$$\otimes : \quad \mathbf{Comp}(H_1) \times \mathbf{Comp}(H_2) \longrightarrow \mathbf{Comp}(H)$$

$$\left(\xrightarrow{\mathsf{In}_1} \boxed{(S_1, \alpha_1)} \xrightarrow{\mathsf{Out}_1}, \xrightarrow{\mathsf{In}_2} \boxed{(S_2, \alpha_2)} \xrightarrow{\mathsf{Out}_2} \right) \mapsto \xrightarrow{\mathsf{In}_1 \times \mathsf{In}_2} \boxed{(S, \alpha)} \xrightarrow{\mathsf{Out}_1 \times \mathsf{Out}_2}$$

as follows: for every component $C_1 = (S_1, init_1, \alpha_1) \in \mathbf{Comp}(H_1)$ *and every component* $C_2 = (S_2, init_2, \alpha_2) \in \mathbf{Comp}(H_2)$, $\otimes((C_1, C_2)) = (S, init, \alpha)$ *where:*

- $S = S_1 \times S_2$ *is the set of states,*
- $init = (init_1, init_2)$ *is the initial state,*
- $\alpha : S \times (\mathsf{In}_1 \times \mathsf{In}_2) \longrightarrow T((\mathsf{Out}_1 \times \mathsf{Out}_2) \times S)$ *is the mapping defined as follows:* $\forall i = (i_1, i_2) \in \mathsf{In}_1 \times \mathsf{In}_2$ *and* $(s_1, s_2) \in S$*:*

$$\alpha((s_1, s_2))(i) = \eta'^{-1}_{(\mathsf{Out}_1 \times \mathsf{Out}_2) \times S} \left\{ ((o_1, o_2), (s_1', s_2')) \mid (o_1, s_1') \in \eta'_{\mathsf{Out}_1 \times S_1}(\alpha_1(s_1)(i_1)) \text{ and } (o_2, s_2') \in \eta'_{\mathsf{Out}_2 \times S_2}(\alpha_2(s_2)(i_2)) \right\}$$

4.1.2. Feedback

The feedback operator is a composition where some outputs of a component are linked to its inputs. This means that some outputs can be fed back as inputs. In order to obtain a model which fits our component definition, we need to take into account the computational effects of the monad T. This monad impacts both the evolution of the state of the component and the observation of its outputs. Therefore, the feedback link between outputs and inputs carries to the inputs part of the structure imposed by T to the outputs. For instance, with the monad built on $\mathcal{P}$ for modeling non-determinism, the feedback may bring non-determinism to the inputs of the component.

We introduce feedback interfaces for defining correspondences between outputs and inputs of components. A feedback interface also allows us to keep only the inputs and the outputs that are not involved in feedback thanks to component-wise projections π_i and π_o:

Definition 4.2 (Feedback interface). *Let* $H = T(\mathsf{Out} \times \_)^{\mathsf{In}}$ *be a signature. A* ***feedback interface*** *over* H *is a triplet* $\mathcal{I} = (f, \pi_i, \pi_o)$ *where:*

- $f : \mathsf{In} \times \mathsf{Out} \longrightarrow \mathsf{In}$ *is a function such that:*

$$\forall (i, o) \in \mathsf{In} \times \mathsf{Out}, f(f(i, o), o) = f(i, o)$$

- $\pi_i : \mathsf{In} \longrightarrow \mathsf{In}'$ *and* $\pi_o : \mathsf{Out} \to \mathsf{Out}'$ *are surjective mappings*[6] *such that:*

$$\forall (i, o) \in \mathsf{In} \times \mathsf{Out}, \pi_i(i) = \pi_i(f((i, o)))$$

The mapping f allows to specify how components are linked and which parts of their interfaces are involved in the composition process. Both mappings π_i and π_o can be thought as extensions of the hiding connective found in process calculi [24]. Thereby, the feedback interface enables encapsulation by making invisible the internal interactions made in the scope of the component. This encapsulation helps to separate both the internal behaviour and local interactions from the external interactions with the global system, and thus to treat interactions between components independently of the behaviour of individual components.

Two kinds of feedback operators are distinguished in this paper: *relaxed* feedback and *synchronous* feedback. The first kind means that in a reaction, the output is not simultaneous with the input. This relaxed feedback composition depends on the previous output and the current input. The second kind means that the reaction of a system takes no observable time [25]. Systems produce their outputs synchronously with their inputs. More precisely, at some reaction r, the output of system $\mathcal{S}$ in r must be available to its inputs in the same reaction r.

[6] *i.e* component-wise projections

Definition 4.3 (Relaxed feedback $\hookleftarrow$). *Let $H = T(\mathsf{Out} \times \_)^{\mathsf{In}}$ be a signature and $\mathcal{I} = (f, \pi_i, \pi_o)$ be a feedback interface over H. Let us note $H' = T(\mathsf{Out}' \times \_)^{\mathsf{In}'}$. Let $C = (S, s_0, \alpha)$ be a component over H. Let us define for every $x \in \mathsf{In}^\omega$, the set Θ_x whose elements are couples $(\bar{x}, y_{\bar{x}}) \in \mathsf{In}^\omega \times \mathsf{Out}^\omega$ inductively defined from an infinite sequence of states $(s_0, s_1, \ldots, s_k, \ldots)$ of S as follows:*

- $\bar{x}(0) = x(0)$ *and* $y_{\bar{x}}(0) \in \eta'_{\mathsf{Out}\times S}(\alpha(s_0)(x(0)))_{|_1}$
- $\forall n, 0 < n < \omega,\ \bar{x}(n) = f(x(n), y_{\bar{x}}(n-1)),\ y_{\bar{x}}(n) \in \eta'_{\mathsf{Out}\times S}(\alpha(s_n)(\bar{x}(n)))_{|_1}$ *and* $s_n \in \eta'_{\mathsf{Out}\times S}(\alpha(s_{n-1})(\bar{x}(n-1)))_{|_2}$

Then, the operation of relaxed feedback over $\mathcal{I}$, $\hookleftarrow_{\mathcal{I}}$: $\mathbf{Comp}(H) \longrightarrow \mathbf{Comp}(H')$ associates to every component $C = (S, s_0, \alpha)$ over H, the component $(\langle\mathbb{F}\rangle, \mathbb{F}, \alpha_{\langle\mathbb{F}\rangle})$ over H' where $\mathbb{F}$ is the set of transfer functions $\mathcal{F} : \mathsf{In}'^\omega \longrightarrow \mathsf{Out}'^\omega$, each one defined by $\mathcal{F}(x') = y'$ where there exists $x \in \mathsf{In}^\omega$ such that there exists $(\bar{x}, y_{\bar{x}}) \in \Theta_x$ satisfying $\forall i < \omega, x'(i) = \pi_i(\bar{x}(i))$ and $y'(i) = \pi_o(y_{\bar{x}}(i))$.

Let us now explain the last definition. We want to build a component that hides the feedback of a component C. As one can see on Figure 3, the feedback component $\hookleftarrow(C)$ is given as a set of transfer functions, each one mapping an infinite sequence of inputs $x' \in \mathsf{In}'^\omega$ to an infinite sequence of outputs $y' \in \mathsf{Out}'^\omega$. The outputs are then hidden from any state s that are fed back as inputs to the successor of s. The result is a component with input and output sets In' and Out' respectively. This is done by means of the feedback interface $\mathcal{I} = (f, \pi_i, \pi_o)$. Let us suppose that the current state of C at the n^{th} reaction is $s_n \in S$ and the current external input is $x(n) \in \mathsf{In}$. Then, let us compute both new input $x'(n) \in \mathsf{In}'$ and output $y'(n) \in \mathsf{Out}'$ when C is triggered by $x(n)$. First, by f, we compute the input $\bar{x}(n) = f(x(n), y_{\bar{x}(n-1)})$. Then, $\bar{x}(n)$ becomes the new input of C. Indeed, component C reacts by updating its state to s_{n+1} and producing an output $y_{\bar{x}(n)}$. In this way, the output of C at the n^{th} reaction is given by relying on the previous output $y_{\bar{x}(n-1)}$ and the current input $x(n)$. Second, by means of π_i and π_o, we hide both input and output involved in the feedback, and then produce the input $x'(n) = \pi_i(\bar{x}(n))$ and the output $y'(n) = \pi_o(\bar{y}(n))$ of the relaxed feedback component $\hookleftarrow_{\mathcal{I}}(C)$.

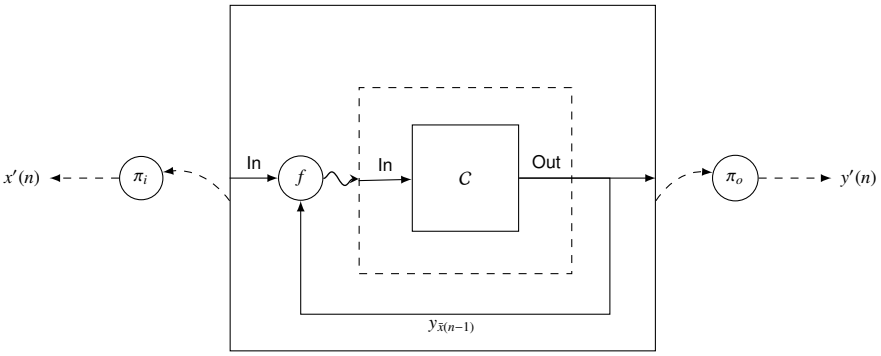


Figure 3: Relaxed feedback composite: $\hookleftarrow(C)$

Proposition 4.1. $\hookleftarrow_{\mathcal{I}}$: $\mathbf{Comp}(H) \longrightarrow \mathbf{Comp}(H')$ *is a functor.*

Proof . *We have to make a correspondence between homomorphisms in* $\mathbf{Comp}(H)$ *and in* $\mathbf{Comp}(H')$.

Let $f : C_1 \longrightarrow C_2$ be an homomorphism in $\mathbf{Comp}(H)$. *Then, let us define $\hookleftarrow_{\mathcal{I}}(f) : \hookleftarrow_{\mathcal{I}}(C_1) \longrightarrow \hookleftarrow_{\mathcal{I}}(C_2)$ where $\hookleftarrow_{\mathcal{I}}(C_i) = (\langle\mathbb{F}_i\rangle, \mathbb{F}_i, \alpha_{\langle\mathbb{F}_i\rangle})$ for $i = 1, 2$ as follows:*

- $\hookleftarrow_{\mathcal{I}}(f)(\mathbb{F}_1) = \mathbb{F}_2$
- *for every $j, 0 < j < \omega$, for every $\mathbb{G}' \in \langle\mathbb{F}_1\rangle^j$, we know by definition that there exists $\mathbb{G} \in \langle\mathbb{F}_1\rangle^{j-1}$, $i \in \mathsf{In}$ and $o \in \mathsf{Out}$ such that:*
  - $o \in \bigcup_{\mathcal{F}\in\mathbb{G}} (\mathcal{F}(i.x)(0))$

– $\mathbb{G}' = \{\mathcal{F}(i.x)' \mid \mathcal{F}(i.x)(0) = o \text{ and } \mathcal{F} \in \mathbb{G}\}$

for $x \in \mathsf{In}^\omega$ chosen arbitrarily. It is sufficient to write down

$$\hookleftarrow_{\mathcal{I}}(f)(\mathbb{G}') = \left\{\mathcal{F}'(i.x)' \mid \mathcal{F}'(i.x)(0) = o \text{ and } \mathcal{F}' \in \hookleftarrow_{\mathcal{I}}(f)(\mathbb{G})\right\}$$

f being a morphism on coalgebras, we can easily show that $\hookleftarrow_{\mathcal{I}}(f)(\mathbb{G}')$ is nonempty.

Let us finish by showing that $\hookleftarrow_{\mathcal{I}}(f)$ preserves identities and compositions. For identities, let $C \in \mathbf{Comp}(H)$, $\hookleftarrow_{\mathcal{I}}(C) = \langle\mathbb{F}\rangle$, and let us prove by induction on the structure of $\mathbb{F}$ that $\hookleftarrow_{\mathcal{I}}(\mathsf{id}_C) = \mathsf{id}_{\hookleftarrow_{\mathcal{I}}(C)}$.

- ***Basic Step:*** *By definition of $\hookleftarrow_{\mathcal{I}}(\mathsf{id}_C)$, one has $\hookleftarrow_{\mathcal{I}}(\mathsf{id}_C)(\mathbb{F}) = \mathbb{F} = \mathsf{id}_{\hookleftarrow_{\mathcal{I}}(C)}(\mathbb{F})$*
- ***Induction Step:*** *let $\mathbb{G}' \in \langle\mathbb{F}\rangle^{j+1}$. We know by definition of $\mathbb{G}'$ that there exists $\mathbb{G} \in \langle\mathbb{F}\rangle^{j}$, $i \in \mathsf{In}$ and $o \in \mathsf{Out}$ such that $o \in \bigcup_{\mathcal{F}\in\mathbb{G}} (\mathcal{F}(i.x)(0))$ and $\mathbb{G}' = \{\mathcal{F}(i.x)' \mid \mathcal{F}(i.x)(0) = o \text{ and } \mathcal{F} \in \mathbb{G}\}$ for $x \in \mathsf{In}^\omega$ chosen arbitrarily. Then, by definition of $\hookleftarrow_{\mathcal{I}}(\mathsf{id}_C)$ one has:*

$$\begin{aligned}
\hookleftarrow_{\mathcal{I}}(\mathsf{id}_C)(\mathbb{G}') &= \left\{\mathcal{F}'(i.x)' \mid \mathcal{F}'(i.x)(0) = o \text{ and } \mathcal{F}' \in \hookleftarrow_{\mathcal{I}}(\mathsf{id}_C)(\mathbb{G})\right\} \text{ by induction hypothesis} \\
&= \left\{\mathcal{F}'(i.x)' \mid \mathcal{F}'(i.x)(0) = o \text{ and } \mathcal{F}' \in \mathsf{id}_{\hookleftarrow_{\mathcal{I}}(C)}(\mathbb{G})\right\} \text{ by definition of } \mathsf{id}_{\hookleftarrow_{\mathcal{I}}(C)} \\
&= \left\{\mathcal{F}'(i.x)' \mid \mathcal{F}'(i.x)(0) = o \text{ and } \mathcal{F}' \in \mathbb{G}\right\} \text{ by hypothesis} \\
&= \mathbb{G}' \\
&= \mathsf{id}_{\hookleftarrow_{\mathcal{I}}(C)}(\mathbb{G}')
\end{aligned}$$

For preservation of composition. Let $f_1 : C_1 \longrightarrow C_2$ and $f_2 : C_2 \longrightarrow C_3$ be two homomorphisms in $\mathbf{Comp}(H)$. Let $\hookleftarrow_{\mathcal{I}}(f_1) : \langle\mathbb{F}_1\rangle \longrightarrow \langle\mathbb{F}_2\rangle$ and $\hookleftarrow_{\mathcal{I}}(f_2) : \langle\mathbb{F}_2\rangle \longrightarrow \langle\mathbb{F}_3\rangle$ their associated homomorphisms in $\mathbf{Comp}(H')$ where $\hookleftarrow_{\mathcal{I}}(C_1) = \langle\mathbb{F}_1\rangle$, $\hookleftarrow_{\mathcal{I}}(C_2) = \langle\mathbb{F}_2\rangle$ and $\hookleftarrow_{\mathcal{I}}(C_3) = \langle\mathbb{F}_3\rangle$.
Let us then prove by induction on the structure of $\mathbb{F}$ that $\hookleftarrow_{\mathcal{I}}(f_2 \circ f_1) = \hookleftarrow_{\mathcal{I}}(f_2) \circ \hookleftarrow_{\mathcal{I}}(f_1)$.

- ***Basic Step:*** *By definition of $\hookleftarrow_{\mathcal{I}}(f_2 \circ f_1)$, one has*

$$\begin{aligned}
\hookleftarrow_{\mathcal{I}}(f_2 \circ f_1)(\mathbb{F}_1) &= \mathbb{F}_3 \text{ by definition of } \hookleftarrow_{\mathcal{I}}(f_2) \\
&= \hookleftarrow_{\mathcal{I}}(f_2)(\mathbb{F}_2) \quad \text{by definition of } \hookleftarrow_{\mathcal{I}}(f_1) \\
&= \hookleftarrow_{\mathcal{I}}(f_2)(\hookleftarrow_{\mathcal{I}}(f_1)(\mathbb{F}_1)) \\
&= \hookleftarrow_{\mathcal{I}}(f_2) \circ \hookleftarrow_{\mathcal{I}}(f_1)(\mathbb{F}_1)
\end{aligned}$$

- ***Induction Step:*** *let $\mathbb{G}'_1 \in \langle\mathbb{F}_1\rangle^{j+1}$. We know by definition of $\mathbb{G}'_1$ that there exists $\mathbb{G}_1 \in \langle\mathbb{F}_1\rangle^{j}$, $i \in \mathsf{In}$ and $o \in \mathsf{Out}$ such that $o \in \bigcup_{\mathcal{F}_1\in\mathbb{G}_1} (\mathcal{F}_1(i.x)(0))$ and $\mathbb{G}'_1 = \{\mathcal{F}_1(i.x)' \mid \mathcal{F}_1(i.x)(0) = o \text{ and } \mathcal{F}_1 \in \mathbb{G}_1\}$ for $x \in \mathsf{In}^\omega$ chosen arbitrarily. By definition of $\hookleftarrow_{\mathcal{I}}(f_1)$, we also know that $\hookleftarrow_{\mathcal{I}}(f_1)(\mathbb{G}'_1) = \left\{\mathcal{F}'_1(i.x)' \mid \mathcal{F}'_1(i.x)(0) = o \text{ and } \mathcal{F}'_1 \in \hookleftarrow_{\mathcal{I}}(f_1)(\mathbb{G}_1)\right\}$.*

  Let us denote by the set $\left\{\mathcal{F}'_1(i_1.x_1)' \mid \mathcal{F}'_1(i_1.x_1)(0) = o_1 \text{ and } \mathcal{F}'_1 \in \hookleftarrow_{\mathcal{I}}(f_1)(\mathbb{G}_1)\right\}$ by $\mathbb{G}'_2$. This set belongs to $\langle\mathbb{F}_2\rangle$. Then, we know by definition of $\mathbb{G}'_2$ that there exists $\mathbb{G}_2 \in \langle\mathbb{F}_2\rangle$ such that $\mathbb{G}_2 = \hookleftarrow_{\mathcal{I}}(f_1)(\mathbb{G}_1)$, $o \in \bigcup_{\mathcal{F}_2\in\mathbb{G}_2} (\mathcal{F}_2(i.x)(0))$ and $\mathbb{G}'_2 = \{\mathcal{F}_2(i.x)' \mid \mathcal{F}_1(i.x)(0) = o \text{ and } \mathcal{F}_2 \in \mathbb{G}_2\}$. By definition of $\hookleftarrow_{\mathcal{I}}(f_2)$, we know that $\hookleftarrow_{\mathcal{I}}(f_2)(\mathbb{G}'_2) = \left\{\mathcal{F}'_2(i.x)' \mid \mathcal{F}'_2(i.x)(0) = o \text{ and } \mathcal{F}'_2 \in \hookleftarrow_{\mathcal{I}}(f_2)(\mathbb{G}_2)\right\}$.

Now, we have that

$$\begin{aligned}
\hookleftarrow_{\mathcal{I}}(f_2)\circ \hookleftarrow_{\mathcal{I}}(f_1)(\mathbb{G}'_1) &= \hookleftarrow_{\mathcal{I}}(f_2)(\hookleftarrow_{\mathcal{I}}(f_1)(\mathbb{G}'_1)) \\
&= \hookleftarrow_{\mathcal{I}}(f_2)(\mathbb{G}'_2) \\
&= \left\{\mathcal{F}'_2(i.x)' \mid \mathcal{F}'_2(i.x)(0) = o \text{ and } \mathcal{F}'_2 \in \hookleftarrow_{\mathcal{I}}(f_2)(\mathbb{G}_2)\right\} \\
&= \left\{\mathcal{F}'_2(i.x)' \mid \mathcal{F}'_2(i.x)(0) = o \text{ and } \mathcal{F}'_2 \in \hookleftarrow_{\mathcal{I}}(f_2)(\hookleftarrow_{\mathcal{I}}(f_1)(\mathbb{G}_1))\right\} \\
&= \left\{\mathcal{F}'_2(i.x)' \mid \mathcal{F}'_2(i.x)(0) = o \text{ and } \mathcal{F}'_2 \in \hookleftarrow_{\mathcal{I}}(f_2)\circ \hookleftarrow_{\mathcal{I}}(f_1)(\mathbb{G}_1)\right\} \text{ induction hypothesis} \\
&= \left\{\mathcal{F}'_2(i.x)' \mid \mathcal{F}'_2(i.x)(0) = o \text{ and } \mathcal{F}'_2 \in \hookleftarrow_{\mathcal{I}}(f_2 \circ f_1)(\mathbb{G}_1)\right\} \\
&= \hookleftarrow_{\mathcal{I}}(f_2 \circ f_1)(\mathbb{G}'_1)
\end{aligned}$$

End

The synchronous feedback is more difficult to define because it requires the existence of an instantaneous fixpoint (i.e. defined at the same time and not deferred of one unit). This gives rise to the notion of *well-formed feedback composition*.

Definition 4.4 (Well-formed feedback composition). *Let $H = T(\mathsf{Out} \times \_)^{\mathsf{In}}$ be a signature. Let C be a component over H and $\mathcal{I} = (f, \pi_i, \pi_o)$ be a feedback interface over H. We say that the **synchronous feedback composition** of C over $\mathcal{I}$ is **well-formed** if, and only if for every state $s \in S$ and every $x \in \mathsf{In}^\omega$, there exists $y \in \mathsf{Out}^\omega$ such that for every $n < \omega,\ y(n) \in \eta'_{\mathsf{Out}\times S}(\alpha(s)(f(x(n), y(n))))_{|_1}$.*

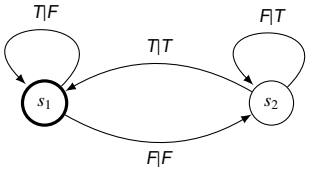
Hence, systems with feedbacks not well-formed will be rejected. They are considered as instable or not well defined systems.

Definition 4.5 (Synchronous feedback $\circlearrowleft$). *Let $H = T(\mathsf{Out} \times \_)^{\mathsf{In}}$ be a signature and $\mathcal{I} = (f, \pi_i, \pi_o)$ be a feedback interface over H. Let us note $H' = T(\mathsf{Out}' \times \_)^{\mathsf{In}'}$. Let us define for every $x \in \mathsf{In}^\omega$, the set Θ_x of output sequences $y \in \mathsf{Out}^\omega$ defined from an infinite sequence of states $(s_0, s_1, \ldots, s_k, \ldots)$ of S as follows:*

$\forall n, 0 \leq n < \omega,\ (y(n), s_{n+1}) \in \eta'_{\mathsf{Out}\times S}(\alpha(s_n)(f(x(n), y(n))))$ (by hypothesis, C's feedback composition being well-formed over $\mathcal{I}$, such y exists)

Then, the operation of synchronous feedback *over $\mathcal{I}$ is the partial mapping $\circlearrowleft_{\mathcal{I}}$: $\mathbf{Comp}(H) \longrightarrow \mathbf{Comp}(H')$ that associates to every component $C = (S, s_0, \alpha)$ over H whose the feedback composition over $\mathcal{I}$ is well-formed, the component $(\langle\mathbb{F}\rangle, \mathbb{F}, \alpha_{\langle\mathbb{F}\rangle})$ over H' where $\mathbb{F}$ is the set of transfer functions $\mathcal{F} : \mathsf{In}'^\omega \longrightarrow \mathsf{Out}'^\omega$, each one defined by $\mathcal{F}(x') = y'$ where there exists $x \in \mathsf{In}^\omega$ s.t. there exists $y \in \Theta_x$ satisfying $\forall i < \omega, x'(i) = \pi_i(x(i))$ and $y'(i) = \pi_o(y_x(i))$.*

Example 4.1. *Consider a component C with state space $S = \{s_1, s_2\}$ and transition function $\alpha : S \times \{T, F\} \longrightarrow \{T, F\} \times S$ defined by the following figure:*



Let us build the composite component that hides the feedback, as suggested by the above definition. For the feedback interface, we choose $f : \mathsf{In} \times \mathsf{Out} \longrightarrow \mathsf{In}$ as the "and" operator, and π_i and π_o as the identities on In and Out respectively. First of all, let us show that the composition is well-formed:

$$\begin{cases} F \in \eta'(\alpha(s_1)(f(F,F)))_{|_1} \\ F \in \eta'(\alpha(s_1)(f(T,F)))_{|_1} \end{cases} \qquad \begin{cases} T \in \eta'(\alpha(s_2)(f(F,T)))_{|_1} \\ T \in \eta'(\alpha(s_2)(f(T,T)))_{|_1} \end{cases}$$

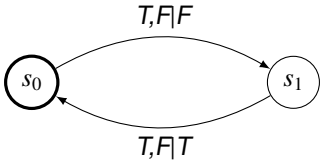
The function $\mathcal{F} : \{T,F\}^\omega \longrightarrow \{F,T\}^\omega$ *defined for every* $x \in \{T,F\}^\omega$ *and for every* $k, 0 \leq k < \omega$, *by:*

$$\mathcal{F}(x)(k) = \begin{cases} F & if\ k\ is\ even \\ T & otherwise \end{cases}$$

is the unique transfer function that can be defined using our synchronous feedback definition. Indeed, both associated outputs to each input $x \in \{T,F\}$ *from* s_1 *are* F, *and both associated outputs to each* $x \in \{T,F\}$ *from* s_2 *are* T. *Then the feedback composite* $\circlearrowleft_{\mathcal{I}}(C)$ *over the interface* $\mathcal{I}$ *is the component* $(\langle\{\mathcal{F}\}\rangle, \{\mathcal{F}\}, \alpha_{\langle\{\mathcal{F}\}\rangle})$ *where the set of states* $\langle\{\mathcal{F}\}\rangle$ *is obtained by a repeated computation of derivative starting from* $\{\mathcal{F}\}$. *The states of* $\langle\{\mathcal{F}\}\rangle$ *then contains the set of all derivative functions of* $\mathcal{F}$ *that are* $\mathcal{F}$ *and* $\mathcal{F}'$ *where* $\mathcal{F}' : \{T,F\}^\omega \longrightarrow \{F,T\}^\omega$ *is the function defined for every* $x \in \{T,F\}^\omega$ *and for every* $k, 0 \leq k < \omega$, *by:*

$$\mathcal{F}'(x)(k) = \begin{cases} T & if\ k\ is\ even \\ F & otherwise \end{cases}$$

This then leads to the following component $\langle\{\mathcal{F}\}\rangle$*:*



Proposition 4.2. $\circlearrowleft_{\mathcal{I}}$: **Comp**$(H) \longrightarrow$ **Comp**(H') *is a partial functor only defined for component* C *whose the synchronous feedback composition over* $\mathcal{I}$ *is well-formed.*

Proof . *The proof is noticeably similar to the proof given for* $\hookleftarrow_{\mathcal{I}}$.
End

We can define as well feedback in terms of its argument as concrete coalgebras as this has been done for product in Definition 4.1, and not on behaviours as this is done in Definitions 4.3 and 4.5. For the synchronous feedback, this leads to:

Definition 4.6 (Synchronous feedback $\circlearrowleft^c$**).** *Let* $H = T(\mathsf{Out} \times \_)^{\mathsf{In}}$ *be a signature and* $\mathcal{I} = (f, \pi_i, \pi_o)$ *be a feedback interface over* H. *Let us note* $H' = T(\mathsf{Out}' \times \_)^{\mathsf{In}'}$. *The* ***operation of synchronous*** [7] ***feedback over*** $\mathcal{I}$ *is the partial functor* $\circlearrowleft^c_{\mathcal{I}}$: **Comp**$(H) \longrightarrow$ **Comp**(H') *that associates to every component* $C = (S, init, \alpha)$ *over* H *for which the feedback composition over* $\mathcal{I}$ *is well-formed, the component* $C' = (S', init', \alpha')$ *over* H' *such that:*

- $S' = S$
- $init' = init$;
- $\alpha' : S' \longrightarrow T(\mathsf{Out}' \times S')^{\mathsf{In}'}$ *is the transition mapping defined by:* $\forall s'_1 \in S', \forall i' \in \mathsf{In}', \alpha'(s'_1)(i') = \eta'^{-1}_{\mathsf{Out}' \times S'}(\Pi)$ *where* Π *is the set:*

$$\Pi = \left\{(o', s'_2) \mid \exists i \in \mathsf{In}, \exists o \in \mathsf{Out}, (o, s'_2) \in \eta'_{\mathsf{Out} \times S}(\alpha(s'_1)(f(i,o))), \pi_i(i) = i' \text{ and } \pi_o(o) = o'\right\}$$

Relaxed feedback can be defined similarly. Definition 4.5 and Definition 4.6 are equivalent. Indeed, it is obvious to check that $\mathsf{beh}_{\circlearrowleft^c_{\mathcal{I}}(C)}(init') = \mathsf{beh}_{\circlearrowleft_{\mathcal{I}}(C)}(\mathbb{F}) = \mathbb{F}$. Although $\circlearrowleft^c_{\mathcal{I}}$ is defined more uniformly with product $\otimes$ because both are defined as concrete coalgebras, the interest of $\circlearrowleft_{\mathcal{I}}$ (resp. $\hookleftarrow_{\mathcal{I}}$) is that the resulting component is the minimal one. This will make compositionality proofs easier in Section 5.2 and Section 8.

[7] The exponent c in $\circlearrowleft^c_{\mathcal{I}}$ is to express that feedback is defined in terms of its argument as concrete coalgebras.

4.2. Complex operators

As explained before, from these basic operators, we can build more complex ones by composition.

Definition 4.7 (Complex operator). *The **set of complex operators**, is inductively defined as follows:*

- $\_$ *is a complex operator of arity* 1*;*
- *if* op_1 *and* op_2 *are complex operators of arity* n_1 *and* n_2 *respectively, then* $op_1 \otimes op_2$ *is a complex operator of arity* $n_1 + n_2$*;*
- *if* op *is a complex operator of arity* n *and* $\mathcal{I}$ *is a feedback interface, then* $\circlearrowleft_{\mathcal{I}}(op)$ *is a complex operator of arity* n*;*
- *if* op *is a complex operator of arity* n *and* $\mathcal{I}$ *is a feedback interface, then* $\hookleftarrow_{\mathcal{I}}(op)$ *is a complex operator of arity* n*.*

In the following, as examples of complex operators, we show how three standard integration operators, respectively sequential composition, synchronous product and concurrent composition, can be defined in our framework.

4.2.1. Sequential composition

The *sequential composition* of two components C_1 and C_2 corresponds to a composition where both components C_1 and C_2 are interconnected side-by-side and the output of one is the input of the other. When C_1 is triggered by an input i from the environment, C_1 processes i and the produced output is fed to C_2 (see Figure 4). A requirement for this composition to be defined is that Out_1 has to be included into In_2 ($\mathsf{Out}_1 \subseteq \mathsf{In}_2$). This ensures that any output produced by C_1 is an acceptable input to C_2.
This kind of composition

$$\begin{array}{ccccc} \rhd : & & \mathbf{Comp}(H_1) \times \mathbf{Comp}(H_2) & \longrightarrow & \mathbf{Comp}(H) \\ & (\xrightarrow{\mathsf{In}_1} \boxed{(S_1, \alpha_1)} \xrightarrow{\mathsf{Out}_1} , \xrightarrow{\mathsf{In}_2} \boxed{(S_2, \alpha_2)} \xrightarrow{\mathsf{Out}_2}) & \mapsto & \xrightarrow{\mathsf{In}_1} \boxed{(S, \alpha)} \xrightarrow{\mathsf{Out}_2} \end{array}$$

can be naturally defined in our framework using both feedback operator and cartesian product by:

$$\rhd((C_1, C_2)) = \ominus_{\mathcal{I}}((C_1 \otimes C_2))$$

where $\mathcal{I} = (f, \pi_i, \pi_o)$ is the feedback interface defined for every $(i, i') \in \mathsf{In}_1 \times \mathsf{In}_2$ and $(o, o') \in \mathsf{Out}_1 \times \mathsf{Out}_2$ as follows:

$$f((i, i'), (o, o')) = (i, o), \quad \pi_i((i, i')) = i \quad \text{and} \quad \pi_o((o, o')) = o'$$

and $\ominus$ stands for $\hookleftarrow$ or $\circlearrowleft$ depending on whether we want a relaxed or instantaneous sequential composition. For the first sequential composition, the output o produced from the component C_1 after triggering by an input i takes some observable time to feed to the component C_2. In this case, $\rhd$ will be denoted by $\rhd_r$. For the second one, the output o produced from the component C_1 after triggering by an input i is directly fed to the input of the component C_2. In this case, $\rhd$ will be denoted by $\rhd_s$.

4.2.2. Synchronous product

The *synchronous product* corresponds to a composition where both components C_1 and C_2 are executed independently or jointly, depending on the input. Hence, C_1 and C_2 are simultaneously executed when triggered by a joint input i that belongs to both inputs set of C_1 and C_2.

This kind of product can also be naturally expressed in terms of the synchronous feedback operator and the cartesian product (see Figure 5). But before, we need to impose that both input and output sets of C_1 and C_2 contain a special input abs to allow components to stutter, i.e. to indicate that no progress of the component is done. The synchronous product is then defined as follows:

$$\circledast((C_1, C_2)) = \rhd_s(C_0, (C_1 \otimes C_2))$$

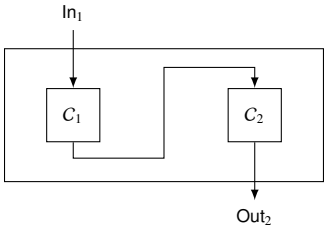


Figure 4: Sequential composition

where $C_0 = (\{init_0\}, init_0, \alpha_0)$ is the component over the signature $T((\mathsf{In}_1 \times \mathsf{In}_2) \times \_)^{\mathsf{In}_1 \cup \mathsf{In}_2}$ where α_0 is the transition mapping defined by: $\forall i \in \mathsf{In}_1 \cup \mathsf{In}_2$

$$\alpha_0(init_0)(i) = \begin{cases} ((i,i), init_0) & \text{if} \quad i \in \mathsf{In}_1 \cap \mathsf{In}_2 \\ ((i, \mathsf{abs}), init_0) & \text{if} \quad i \in (\mathsf{In}_1 \setminus \mathsf{In}_1 \cap \mathsf{In}_2) \\ ((\mathsf{abs}, i), init_0) & \text{otherwise} \end{cases}$$

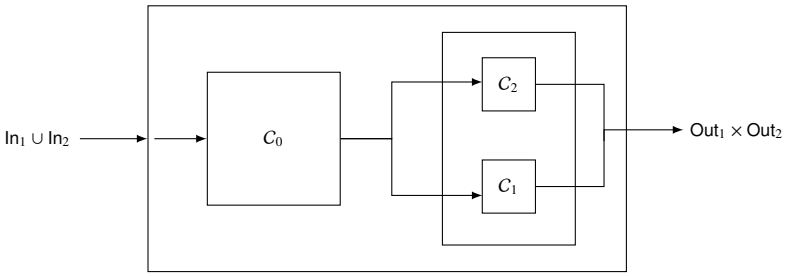


Figure 5: Synchronous product

4.2.3. Concurrent composition

The *concurrent composition*, denoted by $C = \oplus((C_1, C_2))$, of two components C_1 and C_2 corresponds to a composition where both components C_1 and C_2 are executed independently or jointly, depending on the input received from environment. It combines both choice and parallel compositions, in the sense that C_1 and C_2 can be simultaneously executed when triggered by a pair of inputs (i_1, i_2) (i_1 belongs to inputs set of C_1 and i_2 belongs to inputs set of C_2), or separately when triggered by an input i: if $i \in \mathsf{In}_1$, then C_1 is executed and the reaction of C is that of C_1, otherwise C_2 is executed and the reaction of C is that of C_2.

This kind of composition can also be naturally expressed in terms of the synchronous feedback operator and the cartesian product (see Figure 6) as[8] follows:

$$\oplus((C_1, C_2)) = \rhd_s(\rhd_s(C_0, (C_1 \otimes C_2)), C_0') \tag{4}$$

where $C_0 = (\{init_0\}, init_0, \alpha_0)$ is the component over the signature $T((\mathsf{In}_1 \times \mathsf{In}_2) \times \_)^{\mathsf{In}_1 \cup \mathsf{In}_2 \cup \mathsf{In}_1 \times \mathsf{In}_2}$ where α_0 is the transition mapping defined by: $\forall i \in \mathsf{In}_1 \cup \mathsf{In}_2 \cup \mathsf{In}_1 \times \mathsf{In}_2$

$$\alpha_0(init_0)(i) = \begin{cases} (i, init_0) & \text{if} \quad i \in \mathsf{In}_1 \times \mathsf{In}_2 \\ ((i, \mathsf{abs}), init_0) & \text{if} \quad i \in \mathsf{In}_1 \\ ((\mathsf{abs}, i), init_0) & \text{otherwise} \end{cases}$$

[8] Here also we need to impose that $\mathsf{abs} \in \mathsf{In}_1 \cap \mathsf{In}_2$.

and $C'_0 = (\{init'_0\}, init'_0, \alpha'_0)$ is the component over the signature $T((\mathsf{Out}_1 \cup \mathsf{Out}_2 \cup \mathsf{Out}_1 \times \mathsf{Out}_2) \times \_)^{\mathsf{Out}_1 \times \mathsf{Out}_2}$ where α'_0 is the transition mapping defined by: $\forall o = (o_1, o_2) \in \mathsf{Out}_1 \times \mathsf{Out}_2$

$$\alpha_{0'}(init'_0)(o) = \begin{cases} (o_1, init'_0) & \text{if} \quad o \in \mathsf{Out}_1 \times \{\mathsf{abs}\} \\ (o_2, init'_0) & \text{if} \quad o \in \{\mathsf{abs}\} \times \mathsf{Out}_2 \\ (o, init'_0) & \text{otherwise} \end{cases}$$

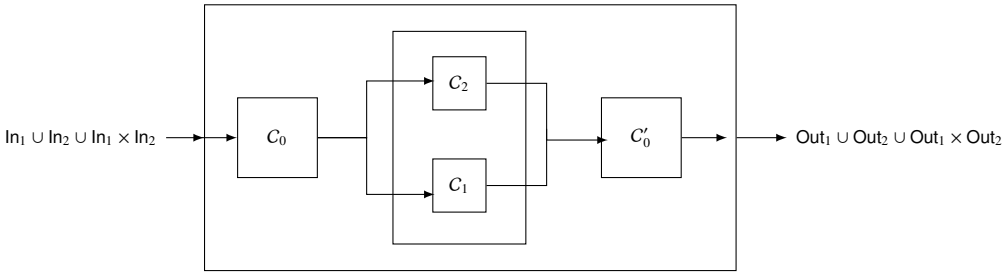


Figure 6: Concurrent composition: $\oplus((C_1, C_2)) = \rhd_s(\rhd_s(C_0, (C_1 \otimes C_2)), C'_0)$

5. Systems and compositionality

5.1. Systems

Complex operators for basic components yield larger components that we will call *systems*. However, it is not always possible to yield a component for a complex operator from any set of basic components passed as arguments. Indeed, for a complex operator of the form $\circlearrowleft_{\mathcal{I}}(op)$, according to the component C resulting from the evaluation of op, the interface $\mathcal{I}$ has to be defined over the signature of C and the feedback over C has to be well-formed over $\mathcal{I}$. This leads up to the following definition:

Definition 5.1 (Systems). *Let $\mathbb{C}$ be a set of components. The **set of systems over** $\mathbb{C}$ is inductively defined as follows:*

- *for any $C \in \mathbb{C}$, a component over a signature H, $\_(C) = C$ is a system over the signature H and $\_$ is **defined for** C;*
- *if $op_1 \otimes op_2$ is a complex operator of arity $n = n_1 + n_2$ then for every sequence $(C_1, C_2, \ldots, C_{n_1}, C_{n_1+1}, \ldots, C_n)$ of components in $\mathbb{C}$ with each C_i over $H_i = T(O_i \times \_)^{I_i}$, if both op_1 and op_2 are defined for $C_1, C_2, \ldots, C_{n_1}$ and $C_{n_1+1}, \ldots, C_n$ respectively, then $op_1 \otimes op_2(C_1, \ldots, C_n) = op_1(C_1, \ldots, C_{n_1}) \otimes op_2(C_{n_1+1}, \ldots, C_n)$ is a system over $H = T(\prod_{i=1}^{n} O_i \times \_)^{\prod_{i=1}^{n} I_i}$ and $op_1 \otimes op_2$ is **defined for** $(C_1, \ldots, C_n)$, else $op_1 \otimes op_2$ is **undefined for** $(C_1, \ldots, C_n)$;*
- *if $\circlearrowleft_{\mathcal{I}}(op)$ is a complex operator of arity n, then for every sequence $(C_1, \ldots, C_n)$ of components in $\mathbb{C}$, if op is defined for $(C_1, \ldots, C_n)$ with $\mathcal{S} = op(C_1, \ldots, C_n)$ is over H, $\mathcal{I}$ is a feedback interface over H and the feedback composition of $\mathcal{S}$ is well-formed, then $\circlearrowleft_{\mathcal{I}}(op)(C_1, \ldots, C_n) = \circlearrowleft_{\mathcal{I}}(\mathcal{S})$ is a system over H' and[9] $\circlearrowleft_{\mathcal{I}}(op)$ is **defined for** $(C_1, \ldots, C_n)$, else $\circlearrowleft_{\mathcal{I}}(op)$ is **undefined for** $(C_1, \ldots, C_n)$.*
- *if $\hookleftarrow_{\mathcal{I}}(op)$ is a complex operator of arity n, then for every sequence $(C_1, \ldots, C_n)$ of components in $\mathbb{C}$, if op is defined for $(C_1, \ldots, C_n)$ with $\mathcal{S} = op(C_1, \ldots, C_n)$ is over H and $\mathcal{I}$ is a feedback interface over H, then $\hookleftarrow_{\mathcal{I}}(op)(C_1, \ldots, C_n) = \hookleftarrow_{\mathcal{I}}(\mathcal{S})$ is a system over H' and[10] $\hookleftarrow_{\mathcal{I}}(op)$ is **defined for** $(C_1, \ldots, C_n)$, else $\hookleftarrow_{\mathcal{I}}(op)$ is **undefined for** $(C_1, \ldots, C_n)$.*

[9] H' is the signature of the synchronous feedback.
[10] H' is the signature of the relaxed feedback.

From Proposition 4.1 and Proposition 4.2, it is not difficult to see that any complex operator op of arity n defines a partial functor from $\mathbf{Comp}(H_1) \times \ldots \times \mathbf{Comp}(H_n) \longrightarrow \mathbf{Comp}(H)$.

Example 5.1. *An encoder/decoder is usually used to guarantee certain characteristics (for example, error detection) when transmitting data across a link. A simple example of such an encoder/decoder is represented in Figure 7. It consists of two parts:*

- *An encoder that takes in incoming a bit sequence and produces an encoded value which is then transmitted on the link. This encoder is considered as a component $\mathcal{E} = (\{s_0, s_1\}, s_0, \alpha_1)$ where the transition function $\alpha_1 : \{s_0, s_1\} \longrightarrow (\{0,1\} \times \{s_0, s_1\})^{\{0,1\}}$ is graphically shown in the left of Figure 7.*
- *A decoder that takes the values from the link and produces the original value. This decoder is considered as a component $\mathcal{D} = (\{q_0, q_1\}, q_0, \alpha_2)$ where the transition function $\alpha_2 : \{q_0, q_1\} \longrightarrow (\{0,1\} \times \{q_0, q_1\})^{\{0,1\}}$ is graphically shown in the right of Figure 7.*

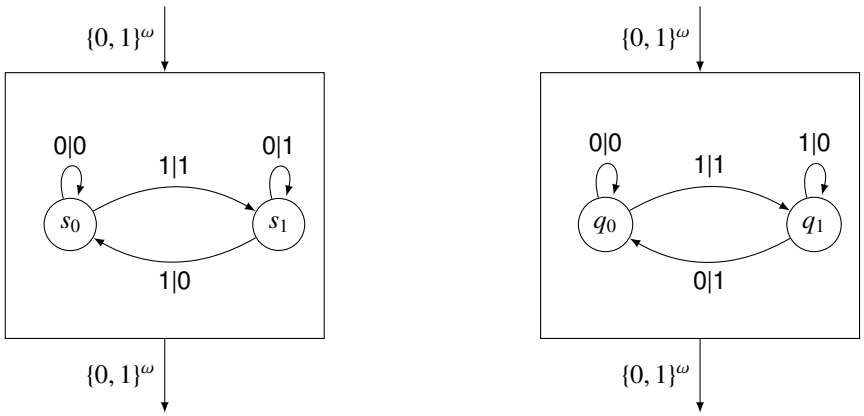


Figure 7: Encoder (on the left) and Decoder (on the right)

Let us now construct the encoder/decoder as a composition of the encoder and the decoder by means of the sequential composition over a synchronous feedback. First of all, let us apply the sequential composition $\rhd_s(\otimes(\mathcal{E}, \mathcal{D}))$ over the synchronous feedback interface $\mathcal{I}$ defined for every $(i, i') \in \mathsf{In}_1 \times \mathsf{In}_2$ and $(o, o') \in \mathsf{Out}_1 \times \mathsf{Out}_2$ by:

$$f((i, i'), (o, o')) = (i, o), \quad \pi_i((i, i')) = i \quad \text{and} \quad \pi_o((o, o')) = o'$$

We first define the cartesian product $C = \otimes(\mathcal{E}, \mathcal{D}))$ of $\mathcal{E}$ and $\mathcal{D}$. It is easy to see that C is well-formed feedback composition over $\mathcal{I}$. Let us check this for (s_0, q_0), we then have:

- $(0,0) \in \eta'(\alpha_C((s_0, q_0))(f((0,0),(0,0))))_{|_1}$
- $(1,1) \in \eta'(\alpha_C((s_0, q_0))(f((1,1),(1,1))))_{|_1}$
- $(0,0) \in \eta'(\alpha_C((s_0, q_0))(f((0,1),(0,0))))_{|_1}$
- $(1,1) \in \eta'(\alpha_C((s_0, q_0))(f((1,0),(1,1))))_{|_1}$

Then, we can apply the synchronous feedback operator $\circlearrowleft_{\mathcal{I}}$ on C. This leads to a minimal component $\langle\{\mathcal{F}\}\rangle$ where $\mathcal{F} : \{0,1\}^\omega \longrightarrow \{0,1\}^\omega$ is the transfer function defined for every $x \in \{0,1\}^\omega$ and every $k, 0 \leq k < \omega$, by $\mathcal{F}(x)(k) = x(k)$.

Let us explain how $\mathcal{F}$ was obtained using a running example. For this, let us consider the bit sequence $(01)^\omega$, and try to find a bit sequence $y \in \{0,1\}^\omega$ that satisfies:

$$\exists (s_0, \ldots, s_k, \ldots) \in S \mid \forall n, 0 \leq n < \omega,\ y(n) \in \eta'_{\mathsf{Out}\times S}(\alpha(s_n)(f(x(n), y(n))))_{|_1}$$

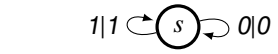
Let us suppose that the current state and the current input are the initial state $s(n) = (s_0, q_0)$ *and* $x(n) = (0,0)$ *respectively. There is a* $y(n) = (0,0)$ *such that*

$$(0,0) \in \eta'_{\mathsf{Out}\times S}(\alpha_C((s_0, q_0))(f((0,0),(0,0))))$$

That is to say, the component C reacts by updating its state to (s_0, q_0) *and producing the output* $(0,0)$*. More precisely, the output of* $\mathcal{E}$ *becomes the input of* $\mathcal{D}$*. So, we can conclude that the input of the encoder/decoder is* $\pi_i(0,0) = 0$ *and its output is* $\pi_o(0,0) = 0$*.*
Suppose next that the current input is $(1,1)$*. Again, there is a* $y(n) = (1,1)$ *such that*

$$(1,1) \in \eta'_{\mathsf{Out}\times S}(\alpha_C((s_0, q_0))(f((1,1),(1,1))))$$

That is to say, the component C reacts by updating its state to (s_1, q_1) *and producing the output* $(1,1)$*. So, we can conclude that the input of the encoder/decoder is* $\pi_i(1,1) = 1$ *and its output is* $\pi_o(1,1) = 1$*.*
Hence, the composite machine alternates states on each reaction and produces the output bit sequence $(01)^\omega$ *for the input bit sequence* $(01)^\omega$*.*
Finally, the minimal component $\langle\{\mathcal{F}\}\rangle$ *that represents* $\mathcal{F}$ *is given by:*



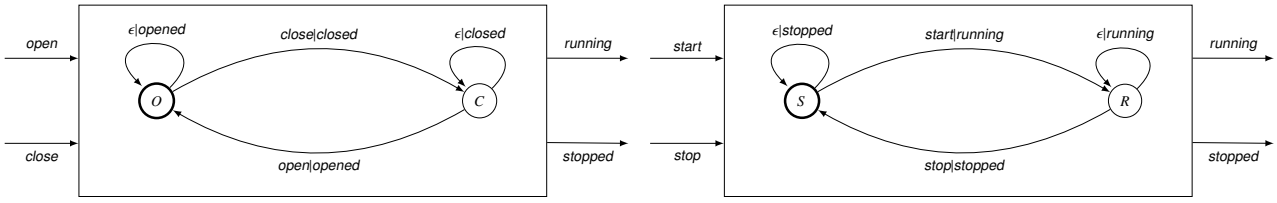
Example 5.2. *We consider a simple microwave* $\mathcal{S}$ *that is built from two basic components: a "door component"* $\mathcal{D}$ *and a "engine component"* $\mathcal{E}$*. In our framework, the door component is defined as* $\mathcal{D} = (\{O, C\}, O, \alpha_{\mathcal{E}})$ *over the signature*

$$(\{\text{opened}, \text{closed}\} \times \_)^{\{\epsilon, \text{open}, \text{close}\}}$$

and the engine component as $\mathcal{M} = (\{S, R\}, S, \alpha_{\mathcal{E}})$ *over the signature*

$$(\{\text{running}, \text{stopped}\} \times \_)^{\{\epsilon, \text{start}, \text{stop}\}}$$

$\alpha_{\mathcal{D}}$ *and* $\alpha_{\mathcal{E}}$ *are depicted in the left and the right sides respectively.*



Let us now show how the microwave system can be obtained by composition the door and the engine components using our basic integration operators. First of all, we apply the cartesian product to $\mathcal{D}$ *and* $\mathcal{E}$*. This leads to a new component* $C = \otimes(\mathcal{D}, \mathcal{E})$ *that is illustrated in Figure 8. We can easily see that:*

- *the microwave cannot run if the door is opened*
- *and opening the door implies the running stop.*

Thus, there is a synchronous feedback that is the output "opened" is returned as an input of the system. Then, we apply the synchronous feedback operator $\circlearrowleft_I^C$ *to C over the signature* $\mathcal{I} = (f, \pi_i, \pi_o)$ *defined by:*

- π_i *as the function that restricts the set* $\{\text{start}, \text{stop}, \epsilon\} \times \{\text{open}, \text{close}, \epsilon\}$ *of inputs of C to the set*

$$\Big\{(\epsilon, \epsilon) \cup \{(i, \epsilon) \mid i \in \{\text{close}, \text{open}\}\} \cup \{(\epsilon, i) \mid i \in \{\text{start}, \text{stop}\}\Big\}$$

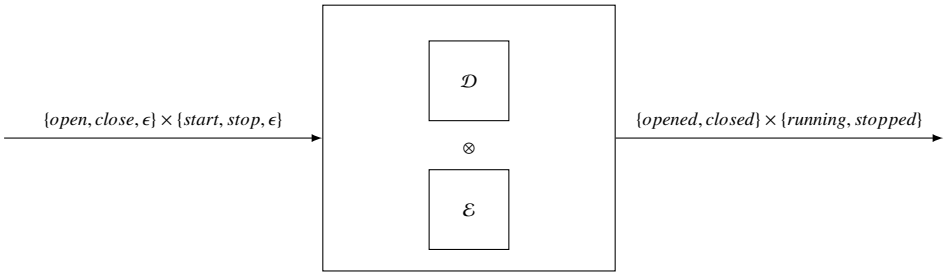


Figure 8: Cartesian product of $\mathcal{E}$ and $\mathcal{D}$

- π_o *as the identity on the set* {running, stopped} × {opened, closed} *of outputs of* C
- f *as the function defined as follows:*
  $f : \Big(\{\text{open, close}, \epsilon\} \times \{\text{start, stop}, \epsilon\}\Big) \times \Big(\{\text{opened, closed}\} \times \{\text{running, stopped}\}\Big) \longrightarrow$
  $\{\text{start} \wedge \neg\text{opened, stop} \vee \text{opened, open, close}\}$

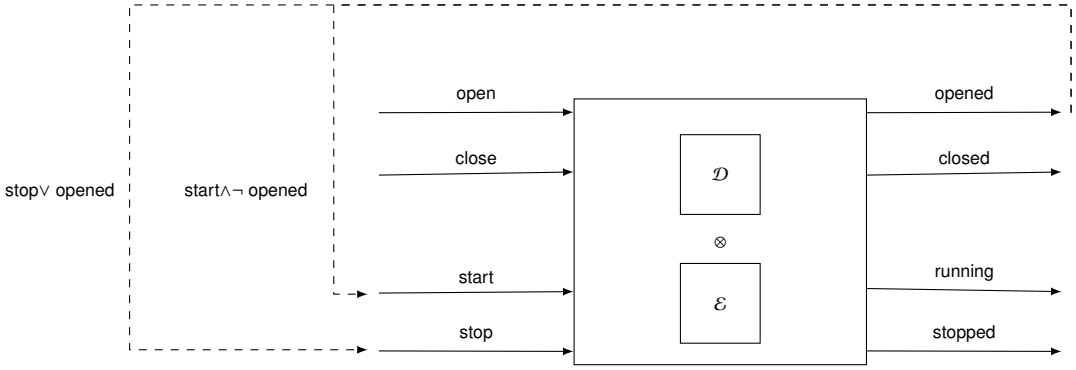


Figure 9: Illustration of the synchronous feedback

This leads to the component[11] *illustrated in Figure 10.*

Let us explain how this component was obtained using a running example. For this, let us consider the infinite input sequence

$$x = ((\text{close}, \epsilon), (\epsilon, \text{start}), (\text{open}, \epsilon))^{\omega}$$

and try to find an infinite output sequence y that satisfies:

$\exists (s_0, \ldots, s_k, \ldots) \in S_C \mid \forall n, 0 \leq n < \omega,\ y(n) \in \eta'(\alpha_C(s_n)(f(x(n), y(n))))_{|_1}$

Let us suppose that the current state and the current input are the initial state $s(n) = (O, S)$ *and* $x(n) = (\text{close}, \epsilon)$ *respectively. There is a* $y(n) = (\text{closed, stopped})$ *such that*

$$(\text{closed, stopped}) \in \eta'(\alpha_C((O, S))(\overbrace{f((\text{close}, \epsilon), (\text{closed, stopped}))}^{(\text{close},\epsilon)}))$$

[11] For the sake of representation simplicity, we preferred to apply the synchronous operator $\circlearrowleft_I^c$ defined in terms of its argument as concrete coalgebras (see Definition 4.6). But, we would have had to apply the synchronous operator $\circlearrowleft_I$.

That is to say, the component C reacts by updating its state to (C, S) *and producing the output* (closed, stopped). *Suppose next that the current input is* (C, S). *Again, for the input* (ϵ, start), *there is a* $y(n) = (\text{closed}, \text{running})$ *such that*

$$(\text{closed}, \text{running}) \in \eta'(\alpha_C((C, S))(\overbrace{f((\epsilon, \text{start}), (\text{closed}, \text{running}))}^{(\epsilon,\text{start})}))$$

That is to say, the component C reacts by updating its state to (C, R) *and producing the output* (closed, running). *Finally, suppose that the current input is* (C, R). *Again, for the input* (open, ϵ), *there is a* $y(n) = (\text{opened}, \text{stopped})$ *such that*

$$(\text{opened}, \text{stopped}) \in \eta'(\alpha_C((C, R))(\overbrace{f((\text{open}, \epsilon), (\text{opened}, \text{stopped}))}^{(\text{open},\text{opened})}))$$

That is to say, the output opened *of* $\mathcal{D}$ *is fed back to the component C and yields an new input* (open, opened). *Hence, the component C reacts by updating its state to* (O, S) *and producing the output* (opened, stopped).

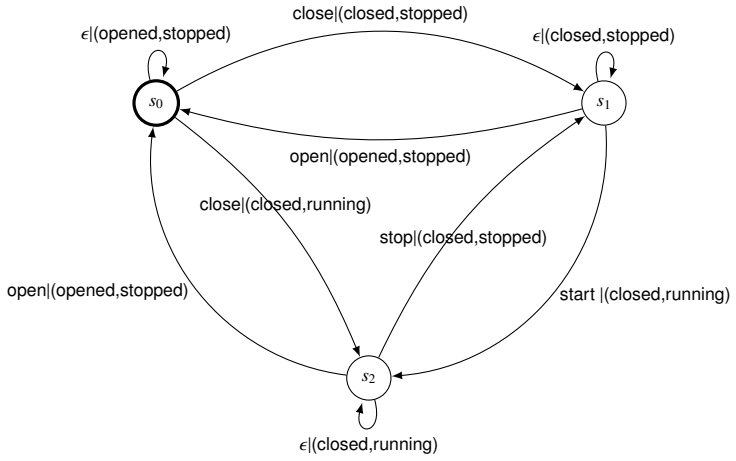


Figure 10: Microwave system

5.2. Compositionality

An important question we wonder about, is compositionality, i.e. is the behaviour of a system the composition of its components behaviours? In our framework, this will be expressed as follows: let op be a complex operator of arity n, $C_1, \ldots, C_n$ be n components and $C = op(C_1, \ldots, C_n)$, then

$$\mathsf{beh}_C(init) = \overline{op}(\mathsf{beh}_{C_1}(init_1), \ldots, \mathsf{beh}_{C_n}(init_n)) \tag{5}$$

where $init$ (resp. $init_i, i = 1, \ldots, n$) is the initial state of C (resp. C_i) and $\overline{op}$ is the adaptation of op on sets of transfer functions. Before establishing Equation 5, we first need to define complex operators $\overline{op}$ on behaviours. Components' behaviours being sets of transfer functions, $\overline{op}$ has to be defined on set of transfer functions. Moreover, it has to respect the same induction structure than op. We have then first to adapt cartesian product and feedback on components' behaviours.

Definition 5.2 (Cartesian product on behaviours $\otimes_f$). *Let* $H_1 = T(\mathsf{Out}_1 \times \_)^{\mathsf{In}_1}$ *and* $H_2 = T(\mathsf{Out}_2 \times \_)^{\mathsf{In}_2}$ *be two signatures. Let* Γ_1 *and* Γ_2 *be two sets of transfer functions over* H_1 *and* H_2 *respectively. Then,* $\Gamma_1 \otimes_f \Gamma_2$ *is the set:*

$$\Gamma_1 \otimes_f \Gamma_2 = \{\mathcal{F}_1 \times \mathcal{F}_2 \mid \mathcal{F}_1 : {\mathsf{In}_1}^\omega \longrightarrow {\mathsf{Out}_1}^\omega, \mathcal{F}_2 : {\mathsf{In}_2}^\omega \longrightarrow {\mathsf{Out}_2}^\omega\}$$

It is obvious to prove that the cartesian product of two transfer functions is a transfer function.

Definition 5.3 (Relaxed feedback on transfer function). *Let $H = T(\mathsf{Out} \times \_)^{\mathsf{In}}$ be a signature and $\mathcal{I} = (f, \pi_i, \pi_o)$ be a feedback interface over H. Let $\mathcal{F} : \mathsf{In}^\omega \longrightarrow \mathsf{Out}^\omega$ be a transfer function. Let us define for every $x \in \mathsf{In}^\omega$, the couple $(\hat{x}, y_{\hat{x}}) \in \mathsf{In}^\omega \times \mathsf{Out}^\omega$ by induction on ω as follows:*

- $\hat{x} = x(0)$ *and* $y_{\hat{x}}(0) = \mathcal{F}(x)(0)$
- $\forall n, 0 < n < \omega,\ \hat{x}(n) = f(x(n), y_{\hat{x}}(n-1)),\ y_{\hat{x}}(n) = \mathcal{F}(\overline{x})(n)$ *where* $\overline{x} \in \mathsf{In}^\omega$ *is any dataflow such that* $\forall j \leq n,\ \overline{x}(j) = \hat{x}(j)$.

Then, $\hookleftarrow_{\mathcal{I}_f}(\mathcal{F}) : \mathsf{In}'^\omega \longrightarrow \mathsf{Out}'^\omega$ is the mapping that associates to $x' \in \mathsf{In}'^\omega$, the data flow $y' \in \mathsf{Out}'^\omega$ such that there exists $x \in \mathsf{In}^\omega$ satisfying: $\forall i < \omega,\ x'(i) = \pi_i(\hat{x}(i))$ and $y'(i) = \pi_o(y_{\hat{x}}(i))$.

Let us observe that Definition 5.3 is noticeably similar to Definition 4.3 except that the choice of $y_{\hat{x}}(n)$ is unique in Definition 5.3 because it is given directly by the transfer function $\mathcal{F}$.

$\hookleftarrow_{\mathcal{I}_f}(\mathcal{F})$ needs some conditions on projections π_i and π_o to be a transfer function. Indeed, π_i and π_o are surjective but by no means they are supposed to be injective. This can then question the causality conditions of $\hookleftarrow_{\mathcal{I}_f}(\mathcal{F})$. Imposing to π_i and π_o to be injective would lead to a too strong condition (π_i and π_o would then be bijective) that is seldom satisfied (e.g. the sequential composition defined in Section 4.2.1). Here, we propose a weaker condition that is satisfied by most of integration operators based on feedback (anyway all defined in the paper).

Assumption 1: $\forall x_1, x_2 \in \mathsf{In}^\omega, \forall j, j \leq n,$

$$\pi_i(x_1(j)) = \pi_i(x_2(j)) \Longrightarrow \begin{cases} \pi_o(\mathcal{F}(x_1)(0)) = \pi_o(\mathcal{F}(x_2)(0)) & if\ j = 0 \\ \pi_o(\mathcal{F}(f(x_1(j), \mathcal{F}(\hat{x}_1)(j-1))) = \pi_o(\mathcal{F}(f(x_2(j), \mathcal{F}(\hat{x}_2)(j-1))) & otherwise \end{cases}$$

Proposition 5.1. *$\hookleftarrow_{\mathcal{I}_f}(\mathcal{F}) : \mathsf{In}'^\omega \longrightarrow \mathsf{Out}'^\omega$ is a transfer function.*

Proof. *Let $\mathcal{F} : \mathsf{In}^\omega \longrightarrow \mathsf{Out}^\omega$ be a transfer function over H and $\hookleftarrow_{\mathcal{I}_f}(\mathcal{F}) : \mathsf{In}'^\omega \longrightarrow \mathsf{Out}'^\omega$ be the function defined in Definition 5.3. Let $x'_1, x'_2 \in \mathsf{In}'^\omega$ be two inputs dataflows for $\hookleftarrow_{\mathcal{I}_f}(\mathcal{F})$ and let us prove that if for every $n, 0 \leq n \leq \omega, x'_1(n) = x'_2(n)$, then $\hookleftarrow_{\mathcal{I}_f}(\mathcal{F})(x'_1(n)) = \hookleftarrow_{\mathcal{I}_f}(\mathcal{F})(x'_2(n))$.*
By induction over ω:

- ***Basic Step:*** $n = 0$

  By definition, $x'_1, x'_2 \in \mathsf{In}'^\omega$, then there exists $x_1, x_2 \in \mathsf{In}^\omega$ such that $x'_1(0) = \pi_i(x_1(0))$ and $x'_2(0) = \pi_i(x_2(0))$, and $\hookleftarrow_{\mathcal{I}_f}(\mathcal{F})(x'_1(0)) = \pi_o(\mathcal{F}(x_1(0)))$. By hypothesis, since $x'_1(0) = \pi_i(x_1(0))$ and $x'_2(0) = \pi_i(x_2(0))$, then $\pi_i(x_1(0)) = \pi_i(x_2(0))$. Then, by Assumption 1, we have that $\pi_o(\mathcal{F}(x_1)(0)) = \pi_o(\mathcal{F}(x_2)(0))$. Hence, $\hookleftarrow_{\mathcal{I}_f}(\mathcal{F})(x'_1(0)) = \pi_o(\mathcal{F}(x_2(0)))$ which by definition equals to $\hookleftarrow_{\mathcal{I}_f}(\mathcal{F})(x'_2(0))$.

- ***Induction Step:***

  By definition of $\hookleftarrow_{\mathcal{I}_f}(\mathcal{F})(x'_1(n+1))$, we know there exists $(\hat{x}_1, \mathcal{F}(\hat{x}_1)) \in \mathsf{In}^\omega \times \mathsf{Out}^\omega$ such that $\forall k, 1 \leq k \leq n+1,\ x'_1(k) = \pi_i(\hat{x}_1(k))$ and $\hookleftarrow_{\mathcal{I}_f}(\mathcal{F})(x'_1(k)) = \pi_o(\mathcal{F}(\hat{x}_1)(k))$ where $\hat{x}_1(k) = f(x(k), \mathcal{F}(\hat{x}_1)(k-1))$.

  By definition of $\hookleftarrow_{\mathcal{I}_f}(\mathcal{F})(x'_2(n+1))$, we know there exists $(\hat{x}_2, \mathcal{F}(\hat{x}_2)) \in \mathsf{In}^\omega \times \mathsf{Out}^\omega$ such that $\forall k, 1 \leq k \leq n+1,\ x'_2(k) = \pi_i(\hat{x}_2(k))$ and $\hookleftarrow_{\mathcal{I}_f}(\mathcal{F})(x'_2(k)) = \pi_o(\mathcal{F}(\hat{x}_2)(k))$ where $\hat{x}_2(k) = f(x(k), \mathcal{F}(\hat{x}_2)(k-1))$.

  By hypothesis, we know that $\forall k, 0 \leq k \leq n,\ x'_1(k) = x'_2(k) \Longrightarrow \hookleftarrow_{\mathcal{I}_f}(\mathcal{F})(x'_1(k)) = \hookleftarrow_{\mathcal{I}_f}(\mathcal{F})(x'_2(k))$. It remains to prove that if $x'_1(n+1) = x'_2(n+1)$, then $\hookleftarrow_{\mathcal{I}_f}(\mathcal{F})(x'_1(n+1)) = \hookleftarrow_{\mathcal{I}_f}(\mathcal{F})(x'_2(n+1))$.

  Since $\forall k, 1 \leq k \leq n+1,\ x'_1(k) = \pi_i(\hat{x}_1(k)), x'_2(k) = \pi_i(\hat{x}_2(k))$ and $x'_1(k) = x'_2(k)$, then by Assumption 1, $\forall k, 1 \leq k \leq n+1$. $\pi_o(\mathcal{F}(\hat{x}_1)(n+1)) = \pi_o(\mathcal{F}(\hat{x}_2)(n+1))$. This last result then yield $\hookleftarrow_{\mathcal{I}_f}(\mathcal{F})(x'_1(n+1)) = \hookleftarrow_{\mathcal{I}_f}(\mathcal{F})(x'_2(n+1))$.

End

Definition 5.4 (Well-formed feedback composition for transfer function). *Let $\mathcal{I} = (f, \pi_i, \pi_o)$ be a feedback interface over a signature H. Let $\mathcal{F} : \mathsf{In}^\omega \longrightarrow \mathsf{Out}^\omega$ be a transfer function. The synchronous feedback composition of $\mathcal{F}$ over $\mathcal{I}$ is **well-formed** if, and only if*

$$\forall x \in \mathsf{In}^\omega,\ (\forall n < \omega, \hat{x}(n) = f(x(n), \mathcal{F}(x)(n))) \Longrightarrow \mathcal{F}(\hat{x}) = \mathcal{F}(x)$$

Definition 5.5 (Synchronous feedback for transfer functions). *Let $\mathcal{I} = (f, \pi_i, \pi_o)$ be a feedback interface over a signature H. Let $\mathcal{F} : \mathsf{In}^\omega \longrightarrow \mathsf{Out}^\omega$ be a transfer function. $\circlearrowleft_{\mathcal{I}_f}(\mathcal{F}) : \mathsf{In}'^\omega \longrightarrow \mathsf{Out}'^\omega$ is the mapping that associates to $x' \in \mathsf{In}'^\omega, y' \in \mathsf{Out}'^\omega$ such that there exists $x \in \mathsf{In}^\omega$ satisfying*

$$\forall i < \omega, x'(i) = \pi_i(f(x(i), \mathcal{F}(x)(i)))\ \ and\ \ y'(i) = \pi_o(\mathcal{F}(f(x(i), \mathcal{F}(x)(i))))$$

Similarly to $\hookleftarrow_{\mathcal{I}_f}(\mathcal{F})$, $\circlearrowleft_{\mathcal{I}_f}(\mathcal{F})$ is a transfer function if the following assumption is satisfied by $\mathcal{F}$.

Assumption 2: $\forall x_1, x_2 \in \mathsf{In}^\omega, \forall j, j \leq n$,

$$\pi_i(x_1(j)) = \pi_i(x_2(j)) \Longrightarrow \pi_o(\mathcal{F}(f(x_1(j), \mathcal{F}(x_1)(j))) = \pi_o(\mathcal{F}(f(x_2(j), \mathcal{F}(x_2)(j)))$$

Proposition 5.2. *$\circlearrowleft_{\mathcal{I}_f}$ is a transfer function.*

Proof . *The technical proof is noticeably similar to the proof given for $\hookleftarrow_{\mathcal{I}_f}$.*
End

Definition 5.6 (Feedback on behaviours). *Let Γ be a set of transfer functions over a signature $H = T(\mathsf{Out} \times \_)^{\mathsf{In}}$. Then, $\odot\Gamma$ is the set of transfer functions:*

$$\odot\Gamma = \{\odot\mathcal{F} \mid \mathcal{F} : \mathsf{In}^\omega \longrightarrow \mathsf{Out}^\omega\}$$

where $\odot$ is either $\hookleftarrow_{\mathcal{I}_f}$ or $\circlearrowleft_{\mathcal{I}_f}$.

Complex operators can be easily extended to behaviours by replacing in Definition 4.7, the symbols $\otimes, \hookleftarrow_{\mathcal{I}}$ and $\circlearrowleft_{\mathcal{I}}$ by $\otimes_f, \hookleftarrow_{\mathcal{I}_f}$ and $\circlearrowleft_{\mathcal{I}_f}$, respectively. In the following, given a complex operator on components we will note $\overline{op}$ its equivalent on behaviours.

Similarly, Definition 5.1 can be easily extended to complex operators on behaviours by replacing each component C_i by a set of transfer functions Γ_i, and $\otimes, \hookleftarrow_{\mathcal{I}}$ and $\circlearrowleft_{\mathcal{I}}$ by $\otimes_f, \hookleftarrow_{\mathcal{I}_f}$ and $\circlearrowleft_{\mathcal{I}_f}$, respectively.

Theorem 5.3 (Compositionality). *Let op be a complex operator on components of arity n. Let $C_1, \ldots, C_n$ be n components. If $C = op(C_1, \ldots, C_n)$, then*

$$\mathsf{beh}_C(init) = \overline{op}(\mathsf{beh}_{C_1}(init_1), \ldots, \mathsf{beh}_{C_n}(init_n))$$

Proof . *In order to prove this theorem, we need to prove the following lemmas:*

Lemma 2. *Let C_1 and C_2 be two components over $H_1 = T(\mathsf{Out}_1 \times \_)^{\mathsf{In}_1}$ and $H_2 = T(\mathsf{Out}_2 \times \_)^{\mathsf{In}_2}$. Let $C = \otimes(C_1, C_2)$ be the product component over $H = T((\mathsf{Out}_1 \times \mathsf{Out}_2) \times \_)^{\mathsf{In}_1 \times \mathsf{In}_2}$. Then we have:*

$$\mathsf{beh}_{C_1 \otimes C_2}((init_1, init_2)) = \mathsf{beh}_{C_1}(init_1) \otimes_f \mathsf{beh}_{C_2}(init_2)$$

Proof . *By definition, $\mathsf{beh}_{C_1 \otimes C_2}((init_1, init_2))$ contains all the transfer functions $\mathcal{F} : (\mathsf{In}_1 \times \mathsf{In}_2)^\omega \longrightarrow (\mathsf{Out}_1 \times \mathsf{Out}_2)^\omega$ that associates to every $(x_1, x_2) \in \mathsf{In}_1 \times \mathsf{In}_2$, a $(y_1, y_2) \in \mathsf{Out}_1 \times \mathsf{Out}_2$ such that there exists an infinite sequence $((o_{11}, o_{21}), (s_{11}, s_{21})), \ldots \in (\mathsf{Out}_1 \times \mathsf{Out}_2) \times (S_1 \times S_2)$ satisfying:*

$$\forall j \geq 1, ((o_{1j}, o_{2j}), (s_{1j}, s_{2j})) \in \eta'_{(\mathsf{Out}_1 \times \mathsf{Out}_2) \times (S_1 \times S_2)}(\alpha((s_{1j-1}, s_{2j-1}))(x_1(j-1), x_2(j-1)))$$

with $(s_{10}, s_{20}) = (init_1, init_2)$*, and for every* $k < \omega, y_i(k) = o_i$ *for* $i = 1, 2$.
Hence, for $i = 1, 2$*, there exists an infinite sequence* $(o_{i1}, s_{i1}), \ldots \in \mathsf{Out}_i \times S_i$ *satisfying*

$$\forall j \geq 1, (o_{ij}, s_{ij}) \in \eta'_{\mathsf{Out}_i \times S_i}(\alpha_i(s_{ij-1})(x_i(j-1)))$$

We can then define a transfer function $\mathcal{F}_i : x_i \mapsto y_i$*. Hence* $\mathcal{F} = \mathcal{F}_1 \otimes_f \mathcal{F}_2$ *and then* $\mathcal{F} \in \mathsf{beh}_{C_1}(init_1) \otimes_f \mathsf{beh}_{C_2}(init_2)$.
By following the same reasoning, we can show that given $\mathcal{F}_i \in \mathsf{beh}_{C_i}(init_i), \mathcal{F}_1 \otimes_f \mathcal{F}_2 \in \mathsf{beh}_{C_1 \otimes C_2}((init_1, init_2))$.
End

Lemma 3. *Let* C' *be a component over* $H = T(\mathsf{Out}' \times \_)^{\mathsf{In}'}$ *and* $C = \hookleftarrow_{\mathcal{I}}(C')$ *be a component over* $H = T(\mathsf{Out} \times \_)^{\mathsf{In}}$. *Let* $\mathcal{I} = (f, \pi_i, \pi_o)$ *where* $f : \mathsf{In}' \times \mathsf{Out}' \longrightarrow \mathsf{In}', \pi_i : \mathsf{In}' \longrightarrow \mathsf{In}$ *and* $\pi_o : \mathsf{Out}' \longrightarrow \mathsf{Out}$ *be a feedback interface. Then we have:*

$$\mathsf{beh}_{\hookleftarrow_{\mathcal{I}}(C')}(init) = \hookleftarrow_{\mathcal{I}_f}(\mathsf{beh}_{C'}(init'))$$

where init is the initial state of $C = \hookleftarrow_{\mathcal{I}}(C')$.

Proof . *Le* $\mathcal{F} \in \mathsf{beh}_{\hookleftarrow_{\mathcal{I}}(C')}(init)$*. By definition,* $\mathcal{F}$ *associates to* $x' \in \mathsf{In}^{\omega}$, $y' \in \mathsf{Out}^{\omega}$ *(when such* y' *exists) such that there exists* $x \in \mathsf{In}'^{\omega}$ *and* $(\hat{x}, y_{\hat{x}}) \in \mathsf{In}'^{\omega} \times \mathsf{Out}'^{\omega}$ *satisfying*

$$\forall i < \omega, x'(i) = \pi_i(\hat{x}(i)) \text{ and } y'(i) = \pi_o(y_{\hat{x}}(i))$$

By definition of $\hat{x}$ *and* $y_{\hat{x}}$*, there exists an infinite sequence* $(init', s'_1, \ldots, s'_k, \ldots) \in S'$ *such that:*

- $\hat{x} = x(0)$ *and* $y_{\hat{x}}(0) \in \eta'_{\mathsf{Out}' \times S'}(\alpha'(init')(\hat{x}(0)))$
- $\forall n, 0 < n < \omega,\ \hat{x}(n) = f(x(n), y_{\hat{x}}(n-1)),\ y_{\hat{x}}(n) \in \eta'_{\mathsf{Out}' \times S'}(\alpha'(s'_n)(\hat{x}(n)))$.

Hence, we can extract a transfer function $\mathcal{F}'$ *that associates to* $\hat{x}$, $y_{\hat{x}}$ *such that* $\hookleftarrow_{\mathcal{I}_f}(\mathcal{F}') = \mathcal{F}$*, and then* $\hookleftarrow_{\mathcal{I}_f}(\mathcal{F}') \in \hookleftarrow_{\mathcal{I}_f}(\mathsf{beh}_{C'}(init'))$.
To prove the other inclusion, we can follow the same reasoning.
End

Lemma 4. *Let* C' *be a component over* $H = T(\mathsf{Out}' \times \_)^{\mathsf{In}'}$ *and* $C = \circlearrowleft_{\mathcal{I}}(C')$ *be a component over* $H = T(\mathsf{Out} \times \_)^{\mathsf{In}}$. *Let* $\mathcal{I} = (f, \pi_i, \pi_o)$ *where* $f : \mathsf{In}' \times \mathsf{Out}' \longrightarrow \mathsf{In}', \pi_i : \mathsf{In}' \longrightarrow \mathsf{In}$ *and* $\pi_o : \mathsf{Out}' \longrightarrow \mathsf{Out}$ *be a feedback interface. Then we have:*

$$\mathsf{beh}_{\circlearrowleft_{\mathcal{I}}(C')}(init) = \circlearrowleft_{\mathcal{I}_f}(\mathsf{beh}_{C'}(init'))$$

where init is the initial state of $C = \circlearrowleft_{\mathcal{I}}(C')$.

Proof . *The technical proof is similar to the proof given for* $\hookleftarrow_{\mathcal{I}}$.
End

Now, Theorem 5.3 is proven by induction on the structure of op as follows:

- ***Basic Step:*** *op is of the form* _. *Its equivalent for sets of transfer functions is also defined by* $\_(\Gamma) = \Gamma$. *The conclusion is then obvious.*
- ***Induction Step:*** *Three cases have to be considered*

– $op = \otimes(op_1, op_2)$ *with arity of* op_1 *is* n_1*, arity of* op_2 *is* n_2 *and* $n_1 + n_2 = n$

By induction hypothesis, we have:

(1) $\mathsf{beh}_{op_1(C_1,\ldots,C_{n_1})}(init) = \overline{op}_1(\mathsf{beh}_{C_1}(init_1), \ldots, \mathsf{beh}_{C_{n_1}}(init_{n_1}))$
where init is the initial state of $op_1(C_1, \ldots, C_{n_1})$*.*

(2) $\mathsf{beh}_{op_2(C_{n_1+1},\ldots,C_n)}(init') = \overline{op}_2(\mathsf{beh}_{C_{n_1+1}}(init_{n_1+1}), \ldots, \mathsf{beh}_{C_n}(init_n))$
where $init'$ *is the initial state of* $op_2(C_{n_1+1}, \ldots, C_n)$*.*

and by the definition of both op_1 *and* op_2*, we have*

(3) $op_2(C_{n_1+1}, \ldots, C_n)$ *and* $op_2(C'_{n_1+1}, \ldots, C'_n)$ *are components.*

Then, ((1) + (2) + (3) + *Lemma 2 implies that*
$\mathsf{beh}_{op_1(C_1,\ldots,C_{n_1})\otimes op_2(C_{n_1+1},\ldots,C_n)}((init, init')) =$
$\overline{op}_1(\mathsf{beh}_{C_1}(init_1), \ldots, \mathsf{beh}_{C_{n_1+1}}(init_{n_1})) \otimes_f \overline{op}_2(\mathsf{beh}_{C_{n_1+1}}(init_{n_1+1}), \ldots, \mathsf{beh}_{C_n}(init_n))$

– op *is of the form* $\hookleftarrow_I(op')$ *and is of arity n.*

Let $C_1, \ldots, C_n$ *be n components such that* $C' = op'(C_1, \ldots, C_n)$*. By induction hypothesis,* $\mathsf{beh}_{C'}(init') = \overline{op'}(\mathsf{beh}_{C_1}(init_1), \ldots, \mathsf{beh}_{C_n}(init_n))$*. It remains to prove that* $\mathsf{beh}_{\hookleftarrow_I(C')}(init) = \hookleftarrow_{I_f}(\mathsf{beh}_{C'}(init'))$ *where init is the initial state of* $C = \hookleftarrow_I(C')$*. This last point is naturally proven by Lemma 3.*

– op *is of the form* $\circlearrowleft_I(op')$ *and is of arity n.*

Let $C_1, \ldots, C_n$ *be n components such that* $C' = op'(C_1, \ldots, C_n)$*. By induction hypothesis,* $\mathsf{beh}_{C'}(init') = \overline{op'}(\mathsf{beh}_{C_1}(init_1), \ldots, \mathsf{beh}_{C_n}(init_n))$*. It remains to prove that* $\mathsf{beh}_{\circlearrowleft_I(C')}(init) = \circlearrowleft_{I_f}(\mathsf{beh}_{C'}(init'))$ *where init is the initial state of* $C = \circlearrowleft_I(C')$*. This last point is naturally proven by Lemma 4.*

End

6. Testing of abstract components

6.1. Conformance Relation

In order to be able to compare the behaviours of the implementation and of its specification, we need to consider both as components over a same signature. However, the implementation behaviour 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. The specification Spec of a component is then the formal description of its behaviour given by a coalgebra over a signature $H = T(\mathsf{Out} \times \_)^{\mathsf{In}}$. On the contrary, its implementation SUT (for *System under Test*) is an executable component, which is considered as a black box [26, 27]. We interact with the implementation through its interface, by providing inputs to stimulate it and observing its behaviour 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* [28, 29] require the inclusion of trace sets. The relation *conf* [30] requires that the implementation behaves according to a specification, but allows behaviours on which the specification puts no constrain. The relation *ioco* [15] is similar to *conf*, but distinguishes inputs from outputs. There are many other types of relations [31, 32].

As already indicated, *ioco* as well as *conf* have received much attention by the community of formal testing community. The reason is the objective of conformance testing is mainly to check whether the implementation behaves as required by the specification i.e. to check if the implementation does what it should do. Hence, a conformance relation has to allow implementations not only to do what is specified, but also to do more than what is specified. This requirement of testing conformance is well satisfied by both *conf* and *ioco* contrary to other relations [28, 29, 31, 32] that require to test behaviours that are not in the specification i.e. the implementation does not have the freedom to produce outputs for any input not considered in the specification.

Since we are dealing with components with input and output, we choose *ioco* and extend it to fit our framework. There are several extensions to *ioco* according to both the type of transition system and the aspect considered to be tested. For instance, *sioco* for symbolic transition systems [9], *sioco* for input-output symbolic transition systems [16], *tioco* for timed labeled transition systems [33], *cspio* for *CSP* process algebra [34], *dioco* for distributed systems [35], *uicoco* for hybrid system [36].

Hence, we define the *ioco* relation that we will call here *cioco*[12] in terms of components as defined in Definition 2.3. We make some modifications to the original definition of *ioco* to fit our component definition. That is, instead of considering that after each trace *tr* of a specification Spec, the possible outputs of the corresponding implementation SUT after procesing *tr* is a subset of the possible outputs of Spec, we consider that for all sequences of inputs considered in Spec, SUT does not produce a sequence of outputs that is not allowed by the specification Spec. A specification of a component as well as its implementation model are considered as coalgebras over a signature $H = T(\mathsf{Out} \times \_)^{\mathsf{In}}$.

The formal definition of *cioco* uses the two following definitions:

Definition 6.1 (Component finite traces). *Let $\mathcal{F} \in \mathsf{beh}_C(init)$ be a trace of a component C. Let $n \in \mathbb{N}$. The **finite trace** of length n, noted $\mathcal{F}_{|_n}$, associated to $\mathcal{F}$ is the whole set of the finite sequence $\langle i_0|o_0, \ldots, i_n|o_n\rangle$ such that there exists $x \in \mathsf{In}^\omega$ where for every $j, 0 \leq j \leq n, x(j) = i_j$, and $\mathcal{F}(x(j)) = o_j$.*
Hence, $Trace(C) = \bigcup_{\mathcal{F} \in \mathsf{beh}_C(init)} \bigcup_{n \in \mathbb{N}} \mathcal{F}_{|_n}$ defines the whole set of finite traces over C.

Definition 6.2. *Let C be a component over $T(\mathsf{Out} \times \_)^{\mathsf{In}}$. Let tr be a finite trace of C and $i \in \mathsf{In}$. The set of the possible outputs for the input i after executing tr on C is:*

$$Out(C \text{ after } (tr, i)) = \{o \quad | \quad tr.\langle i|o\rangle \in Trace(C)\}$$

Definition 6.3 (cioco). *Let* Spec *be a component over the signature $T(\mathsf{Out} \times \_)^{\mathsf{In}}$ and* SUT *be its implementation defined as a component*[13] *over $T(\mathsf{Out}' \times \_)^{\mathsf{In}'}$ such that $\mathsf{In} \subseteq \mathsf{In}'$ and $\mathsf{Out} \subseteq \mathsf{Out}'$ and SUT is input-enabled.* SUT *is said* **cioco** Spec*, noted* SUT *cioco* Spec*, if and only if:*

$$\forall tr \in Trace(\mathsf{Spec}), \forall i \in \mathsf{In}, Out(\mathsf{SUT} \text{ after } (tr, i)) \subseteq Out(\mathsf{Spec} \text{ after } (tr, i))$$

6.2. 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 [13, 37] test purposes are described independently of the model of the specification. On the contrary, following [10], 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 FCT 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 $\odot$.

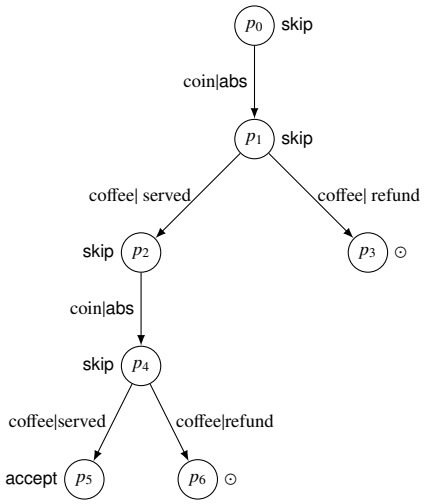
Definition 6.4 (*C*-path). *Let (S, s_0, α) be a component over $T(\mathsf{Out} \times \_)^{\mathsf{In}}$. A **C-path** is defined by two finite sequences of states and inputs $(s_0, \ldots, s_n)$ and $(i_0, \ldots, i_{n-1})$ such that for every $j, 1 \leq j \leq n, s_j \in \eta'_{\mathsf{Out} \times S}\left(\alpha(s_{j-1})(i_{j-1})\right)_{|_2}$.*

Definition 6.5 (Finite computation tree of a component). *Let (S, s_0, α) be a component over $T(\mathsf{Out} \times \_)^{\mathsf{In}}$. The **finite computation tree** of depth n of C, noted $FCT(C, n)$ is the coalgebra $(S_{FCT}, s^0_{FCT}, \alpha_{FCT})$ defined by :*

- *S_{FCT} is the whole set of $C-$*paths

[12] c for component.

[13] Classically in conformance testing, it is assumed that (1) the implementation is input-enabled i.e. it accepts all inputs at all times in order to produce, at any state, answers for all possible inputs providing by the environment. (2) the alphabet of inputs and outputs of Spec and SUT are compatible i.e. $\mathsf{In} \subseteq \mathsf{In}'$ and $\mathsf{Out} \subseteq \mathsf{Out}'$ in order to allow the specification Spec to accept all responses of the implementation SUT.



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

Figure 11: Test purpose of the coffee machine

- s^0_{FCT} *is the initial* $C-$path $\langle s_0, () \rangle$
- α_{FCT} *is the mapping which for every* $C-$path $\langle (s_0, \ldots, s_n), (i_0, \ldots, i_{n-1}) \rangle$ *and every input* $i \in \mathsf{In}$ *associates* $\eta'^{-1}_{\mathsf{Out} \times S_{FCT}}(\Gamma)$ *where* Γ *is the set:*

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

Definition 6.6 (Test Purpose). *Let* $FCT(C, n)$ *be the finite computation tree of depth n associated to a component C. A* ***test purpose*** TP *for C is a mapping* $TP : S_{FCT} \longrightarrow \{\mathsf{accept}, \mathsf{skip}, \odot\}$ *such that:*

- *there exists a* $C-$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:*

  $$\forall j, 1 \leq j \leq n-1, TP(\langle (s_0, \ldots, s_j), (i_0, \ldots, i_{j-1}) \rangle) = \mathsf{skip}$$

- $TP(\langle s_0, () \rangle) = \mathsf{skip}$
- *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 6.1. *Figure 11 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 behaviours 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 behaviour from these nodes.* p_5 *is tagged with* accept *because it is a leaf which corresponds to an expected behaviour. All nodes leading from the root* p_0 *to this node are tagged with* skip *because they are valid prefixes of* p_5*.*

In order to build a test purpose on a finite computation tree, we therefore choose the leaves of the tree that we accept as correct finite behaviours and we tag them with accept. We then tag every node which represents a prefix of an accepted behaviour with skip. The other nodes which lead to behaviours 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.

7. 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 an SUT to its 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 $\odot$, then behaviour is not of interest and the test is said inconclusive (INCONC verdict). If one of the nodes is tagged accept, then the test succeeds (PASS verdict). It may happen, due to the non-determinism of the specification, that the implementation can reach from a given state both an accept state and a $\odot$ state. That means we are not sure to have achieved the test purpose (WeakPASS verdict).

7.1. 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 $[ev_0, ev_1, \ldots, ev_n|V]$, where for all $j \in [0, \ldots, n], ev_j = i|o$ is an elementary input-output with $i \in \mathsf{In} \cup \{\bot\}, o \in \mathsf{Out} \cup \{\bot\}$, and $V \in \{FAIL, PASS, INCONC, WeakPASS\}$. We have added the special symbol $\bot$ to the input and output actions to denote a stimulation of SUT without input and the absence of output for a stimulation. We denote by $stimobs(i|o)$ 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. $NextSkip(CS, ev)$ and $NextPass(CS, ev)$ are the set of states in $Next(CS, ev)$ which are labelled by *skip* and *accept* respectively.

Definition 7.1. *Let $TP : S_{FCT} \rightarrow \{accept, skip, \odot\}$ be a test purpose for a component C, $ev = \langle i|o\rangle$ an event, and S' a subset of S_{FCT}:*

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

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

7.2. 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 satisfied, 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[14]: $\frac{}{\{s^0_{FCT}\}}$

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

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.

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

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

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

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.

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

Example 7.1. *We consider the test purpose TP defined in Figure 11, and show how test cases can be obtained by applying the rules presented in Section 7.2. Let us first recall that the algorithm uses the following notation:*

$$CS \xrightarrow[\textbf{\textit{rule}}]{\textbf{\textit{event}}} CS'$$

where:

- ***event** denotes the current element of the considered trace, and is of the form input|output;*
- ***rule** stands for the rule applied to get the next set of states CS'.*

***FAIL**: To get the verdict $FAIL$, we consider the following trace:*

$$[\text{coin|abs}, \text{coffee|served}, \text{coin|refund} \mid FAIL]$$

The algorithm is applied as follows:

$$\xrightarrow[\textbf{\textit{rule 0}}]{} CS_0 = \{p_0\} \xrightarrow[\textbf{\textit{rule 1}}]{\text{coin|abs}} CS_1 = \{p_1\} \xrightarrow[\textbf{\textit{rule 1}}]{\text{coffee|served}} CS_2 = \{p_2\} \xrightarrow[\textbf{\textit{rule 2}}]{\text{coin|refund}} FAIL$$

The verdict $FAIL$ is due to the following equality:

$$Next(CS_2, \text{coin|refund}) = \emptyset$$

***INCONC**: To get the verdict $INCONC$, we consider the following trace:*

$$[\text{coin|abs}, \text{coffee|served}, \text{coin|abs}, \text{coffee|refund} \mid INCONC]$$

The algorithm is applied as follows:

$$\xrightarrow[\textbf{\textit{rule 0}}]{} CS_0 = \{p_0\} \xrightarrow[\textbf{\textit{rule 1}}]{\text{coin|abs}} CS_1 = \{p_1\} \xrightarrow[\textbf{\textit{rule 1}}]{\text{coffee|served}} CS_2 = \{p_2\} \xrightarrow[\textbf{\textit{rule 1}}]{\text{coin|abs}} CS_3 = \{p_4\} \xrightarrow[\textbf{\textit{rule 3}}]{\text{coffee|refund}} INCONC$$

The verdict $INCONC$ is due to the following two equalities:

- $Next(CS_3, \text{coffee|refund}) = \{p_{11}\} \neq \emptyset$
- $NextPass(CS_3, \text{coffee|refund}) = NextSkip(CS_3, \text{coffee|refund}) = \emptyset$

Pass*: To get the verdict PASS, we consider the following trace:*

$$[\text{coin|abs}, \text{coffee|served}, \text{coin|abs}, \text{coffee|served} \mid PASS]$$

The algorithm is applied as follows:

$$\xrightarrow[\textbf{rule 0}]{} CS_0 = \{p_0\} \xrightarrow[\textbf{rule 1}]{\text{coin|abs}} CS_1 = \{p_1\} \xrightarrow[\textbf{rule 1}]{\text{coffee|served}} CS_2 = \{p_2\} \xrightarrow[\textbf{rule 1}]{\text{coin|abs}} CS_3 = \{p_4\} \xrightarrow[\textbf{rule 4}]{\text{coffee|served}} PASS$$

The verdict $PASS$ is due to the following equality:

$$NextPass(CS_3, \text{coffee|served}) = Next(CS_3, \text{coffee|served}),\ Next(CS_3, \text{coffee|served}) \neq \emptyset$$

WeakPASS *There is no test cases ending by the verdict $WeakPASS$ for TP.*

Let us note here that each of these rules except rule 0 can be used in several ways according to the form of ev. When $ev = \perp|o$, o is produced spontaneously by SUT. When $ev = i|\perp$, 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 ([13, 10, 38]) by choosing the appropriate monad T and input and output sets.

7.3. Properties

A test case informs us about the conformance of the implementation to its specifcation. 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 applied to an implementation under test and its specifcation, 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 7.2 (Execution). *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 $TS(TP, \text{SUT})$ (see its explanation in Section 7.2) is the coalgebra (S_{TS}, α_{TS}) over the signature $(\_)^{\mathbb{EV}}$ defined by :*

- $S_{TS} = \mathbb{CS} \cup \mathbb{V}$ *where $\mathbb{V}$ 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 :*

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

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

Definition 7.3 (Test case). *Let $TS(TP, \mathsf{SUT}) = (S_{TS}, \alpha_{TS})$ be the execution of the test generation algorithm on* SUT *according to TP. A **test case** for TP is a sequence $[ev_0, \ldots, ev_n|V]$ 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 $V \in \mathbb{V}$ such that $V = \alpha_{TS}(s_n)(ev_n)$.*
We note $st(TP, \mathsf{SUT})$ the set of all possible test cases for TP.

We can now introduce the notation:

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

Theorem 7.1. *(Correctness and completeness) For any specification* Spec *and any* SUT*:*

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

Proof .
***Proof of the correctness:** Let* Spec *$= (S, s_0, \alpha)$ be a specification over a signature $H = T(\mathsf{Out} \times \_)^{\mathsf{In}}$ 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:*

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

we have to prove:

$$\text{if there exists a test purpose TP such that FAIL} \in vdt(TP, \mathsf{SUT}), \text{ then}$$

$$\text{SUT does not conform } \textit{w.r.t cioco} \text{ to Spec.}$$

More precisely, according to the definition of cioco, we have to prove that:

$$\text{there exists a finite trace } tr \in Trace(FCT), \text{ an input } i \in \mathsf{In} \text{ such that}$$

$$Out_{\mathsf{SUT}}(\mathsf{SUT} \text{ after } (tr, i)) \nsubseteq Out_{FCT}(FCT \text{ after } (tr, i))$$

This is proved by the following proposition:

***Proposition 7.2.** If there exists a test purpose TP such that $[i_0|o_0, \ldots, i_n|o_n|FAIL] \in st(TP, \mathsf{SUT})$, then:*

1. $\langle i_0|o_0, i_1|o_1, \ldots, i_{n-1}|o_{n-1}\rangle \in Trace(FCT)$.
2. $i_n \in \mathsf{In}$
3. $o_n \in Out_{\mathsf{SUT}}(\mathsf{SUT} \text{ after } (\langle i_0|o_0, \ldots, i_{n-1}|o_{n-1}\rangle, i_n))$.
4. $o_n \notin Out_{\mathsf{Spec}}(\mathsf{Spec} \text{ after } (\langle i_0|o_0, \ldots, i_{n-1}|o_{n-1}\rangle, i_n))$.

First of all, let us denote $\langle i_0|o_0 \ldots i_{n-1}|o_{n-1}\rangle$ by $\langle ev_0 \ldots ev_{n-1}\rangle$.

Proof of (1).
In order to show that the sequence $\langle i_0|o_0 \ldots i_{n-1}|o_{n-1}\rangle \in Trace(FCT)$, we are going to reason on the way of computation of this sequence by using the inference rules. First of all, let $TS(TP, \mathsf{SUT})$ be the execution of the test generation algorithm and $st(TP, \mathsf{SUT})$ be the set of generated test cases. Since $[i_0|o_0, \ldots, i_n|o_n|FAIL] \in st(TP, \mathsf{SUT})$, 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 7.2. Hence, by Definition 7.1, 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. Then, the sequence $\langle ev_0 \ldots ev_j \ldots ev_{n-1}\rangle \in Trace(FCT)$.

Proof of (2).
We have above proved that $\langle i_0|o_0 \dots i_{n-1}|o_{n-1}\rangle \in Trace(FCT)$ *and* $S_n \neq \emptyset$. *We have that* $[i_0|o_0 \dots i_n|o_n|FAIL] \in st(TP, \mathsf{SUT})$ i.e. *submitting the input* i_n *to the implementation under test will produce the output* o_n *that is not specified in* $FCT(C)$. *Then, it is clear that* $i \in \mathsf{In}$ *is an input of* $FCT(\mathsf{Spec})$.

Proof of (3).
It is obvious because $[i_0|o_0, i_1|o_1, \dots, i_n|o_n|FAIL] \in st(TP, \mathsf{SUT})$.

Proof of (4).
We know that $\langle i_0|o_0 \dots i_{n-1}|o_{n-1}\rangle \in Trace(FCT)$ *and* $S_n \neq \emptyset$. *We have that* $[i_0|o_0 \dots i_n|o_n|FAIL] \in st(TP, \mathsf{SUT})$ i.e. *applying* $i_n|o_n$ *have to lead to a* $FAIL$ *verdict. This means that* $\alpha_{TS}(S_n)(i_n|o_n) = FAIL$. *Hence by Definition 7.2,* $Next(S_n, i_n|o_n)$ *has to be empty. But we know that* $Next(S_n, i_n|o_n) \subseteq S_{FCT}$. *Hence,* $\langle i_0|o_0, \dots, i_{n-1}|o_{n-1}, i_n|o_n\rangle$ *does not belong to* $Trace(FCT)$.

Proof of the completeness : *Let* $\mathsf{Spec} = (S, s_0, \alpha)$ *be a specification over a signature* $H = T(\mathsf{Out} \times \_)^{\mathsf{In}}$ *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* $[ev_0, \dots, ev_n|FAIL] \in st(TP, \mathsf{SUT})$.
Since SUT *does not conform to* Spec, *according to the definition of cioco, there exists a trace* $tr = \langle ev_0 \dots ev_{n-1}\rangle \in Trace(FCT)$ *and an input* $i \in \mathsf{In}$ *such that*

$$Out_{\mathsf{SUT}}(\mathsf{SUT} \text{ after } (tr, i)) \nsubseteq Out_{FCT}(FCT \text{ after } (tr, i))$$

i.e. there exists an output o'_n *of* SUT *such that*

- $o'_n \in Out_{\mathsf{SUT}}(\mathsf{SUT} \text{ after } (tr, i_n))$;
- $o'_n \notin Out_{FCT}(FCT \text{ after } (tr, i_n))$.

That means:

$$\langle ev_0, \dots, ev_{n-1}, i_n|o'_n\rangle \in Trace(\mathsf{SUT}) \tag{6}$$

and

$$\langle ev_0, \dots, ev_{n-1}, i_n|o'_n\rangle \notin Trace(FCT) \tag{7}$$

Since $i_n \in \mathsf{In}$, *then there also exists an output* o_n *such that* $o_n \in Out_{FCT}(FCT \text{ after } (tr, i_n))$ *i.e.*

$$\langle ev_0, \dots, ev_{n-1}, i_n|o_n\rangle \in Trace(FCT) \tag{8}$$

Let us denote $\langle i_n|o_n\rangle$ *by* ev_n *and* $\langle i_n|o'_n\rangle$ *by* ev'_n.

Now, let us denote by TP *a test purpose of* FCT *such that there exists a state* $s \in S_{FTC}$ *such that* s *belongs to the set of reachable states from the initial state of* FCT *after executing the trace* $\langle ev_0 \dots ev_{n-1}ev_n\rangle$ *on* FCT, *and* $TP(s) = \mathsf{accept}$ i.e. $\langle ev_0 \dots ev_{n-1}ev_n\rangle$ *forms a path of* TP. *Let us prove that there exists* $[ev_0 \dots ev_{n-1}ev'_n|FAIL] \in st(TP, \mathsf{SUT})$. *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 $\langle ev_0 \dots ev_{n-1}\rangle \in Trace(FCT)$, *then, for every* $j, 0 \leq j < n, S_j$ *exists because for every* $j, 1 \leq j < n, \alpha_{TS}(S_j)(ev_j) = Next(S_j, ev_j)$ *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 Equation 7, we have $\langle ev_0 \dots ev_{n-1}ev'_n\rangle \notin Trace(FCT)$ *and by Equation 6* $\langle ev_0 \dots ev_{n-1}ev_n\rangle \in Trace(\mathsf{SUT})$, *hence* $Next(S_n, ev'_n) = \emptyset$, *and consequently* $\alpha_{TS}(S_n)(ev_n) = FAIL$.
End

8. Compositional testing

Compositional testing provides a technique for checking the correctness of complex components built from simpler correct ones. This means that the correctness of components implies the correctness of systems obtained by assembling them. Several compositional testing approaches have been proposed [17, 39, 40, 41, 42]. These approaches vary according to both formalism and integration operators. In [17], it has been proved that the conformance testing *ioco* based on labeled transition systems is only compositional with respect to parallel composition when specifications and implementations are assumed[15] input-enabled. In [39], the authors extend the testing theory defined in the setting of CSP process algebra whose conformance relation *cspio* is an adapted version of *ioco* to CSP formalism [34], to be able to address testing compositional proposed in [17]. it has been then shown that *cspio* is compositional not only for parallel composition $\|$ but also for other CSP's composition operators by assuming input completeness of the specification in the same alphabet of the implementation. In [40, 41, 42, 43], the authors address differently the compositional testing problem from [17, 39]. In [40], the authors indeed work with input-output symbolic transition systems ($IOSTS$) and propose to test each component of a system in isolation by generating accurate test purposes for them from the global specification of the system and assuming that the specification of every component in the system is available. This allowed them to test the global system by selecting behaviours of basic components that are typically activated in the system, and then re-enforce unitary testing with respect to those behaviours. In [41, 42], the authors study how to design a component when combined with a known part of the system, called the context, has to satisfy a given overall specification in the context of finite state machine. Finally, in [43], the authors extend the so-called *assume-guarantee reasoning* [44] used in model checking areas as a means to cope with the state explosion problem of compositional testing. They then proposed to test each component of a system separately, while taking into account assumptions about the context of the component. They use the input-output labeled transition systems as behavioural models of components and the parallel composition $\|$ to compose components. The conformance relation used in this approach is the *ioco* relation. The underlying idea behind this approach is to check that, given a assumption A about the environment in which the components are supposed to operate, such that iut_2 *ioco* A and ($\mathsf{iut}_1 \| A$) *ioco* spec then ($\mathsf{iut}_1 \| \mathsf{iut}_2$) *ioco* spec. The authors showed that this property holds if the assumption A is input-enabled. This approach then requires the specification spec to be given as a single model rather than a set of components unlike our approach. They do not impose input-completeness of specifications which gives them an advantage with respect to other approaches.

In this contribution, our goal is to extend [17] to our framework, but the other approaches would have been able to be extended in our framework. Besides, this has been done for the approach developed in [40] (cf. [45]).

The underlying idea is then to test an integrated system assuming that its underlying components have already been tested and are correct. The operators used to compose components are supposed to be correctly implemented and to preserve their specifications. Thus, the problem of compositional testing that we address here can be seen as follows: if single components of a system conform to their specifications, what can be said concerning conformance of the whole system according to its specification? This is formally expressed as: $\forall i, 1 \leq i \leq n, I_i$ **rel** S_i implies $op(I_1, \ldots, I_n)$ **rel** $op(S_1, \ldots, S_n)$ where **rel** denotes the conformance relation of interest, $I_1, \ldots, I_n$ are implementation models, $S_1, \ldots, S_n$ are specifications, and **op** is an integration operator. Thus, such a compositional testing theory provides a way to test the integrated system only by testing its sub-systems i.e. there is no need to re-test its conformance correction.

We show here that *cioco* is naturally compositional for the cartesian product. However, compositionality does not hold for *cioco* with respect to the feedback operators, unless the specification model is input-enabled.

Theorem 8.1. *Let* $H_1 = T(\mathsf{Out}_1 \times \_)^{\mathsf{In}_1}$ *and* $H_2 = T(\mathsf{Out}_2 \times \_)^{\mathsf{In}_2}$ *be two signatures. Let* $H = T((\mathsf{Out}_1 \times \mathsf{Out}_2) \times \_)^{\mathsf{In}_1 \times \mathsf{In}_2}$ *be the cartesian product interface for* H_1 *and* H_2*. Let* $I_j, S_j \in$ **Comp**(H_j) *for* $j = 1, 2$ *and* $\otimes((I_1, I_2))$, $\otimes((S_1, S_2)) \in$ **Comp**(H)*. Then, we have:*

$$\left.\begin{array}{l} I_1 \textit{ cioco } S_1 \\ I_2 \textit{ cioco } S_2 \end{array}\right\} \Longrightarrow \otimes((I_1, I_2)) \textit{ cioco } \otimes((S_1, S_2))$$

[15] *input-enabled* means all input actions are always enabled in any state.

Proof . *Let us assume that:*

$$(I_1 \ cioco \ S_1) \text{ and } (I_2 \ cioco \ S_2)$$

and let us then prove that:

$$\otimes((I_1, I_2)) \ cioco \ \otimes((S_1, S_2))$$

Let us use the contradiction principle. For this, let us assume that $\neg(\otimes((I_1, I_2)) \ cioco \ \otimes((S_1, S_2)))$ i.e *that there exists a finite trace* $tr = \langle (i_1, i'_1)|(o_1, o'_1), \ldots, (i_n, i'_n)|(o_n, o'_n)\rangle \in Trace(\otimes((S_1, S_2)))$ *and* $(i, i') \in \mathsf{In}_1 \times \mathsf{In}_2$ *such that there exists an output* $(o, o') \in \mathsf{Out}_1 \times \mathsf{Out}_2$ *among the outputs obtained after executing* $(tr, (i, i'))$ *on* $\otimes((I_1, I_2))$ *not belonging to the ones obtained after executing* $(tr, (i, i'))$ *on* $\otimes((S_1, S_2))$.

Now, we have $tr = \langle (i_1, i'_1)|(o_1, o'_1), \ldots, (i_n, i'_n)|(o_n, o'_n)\rangle \in Trace(\otimes((I_1, I_2)))$. *According to the definition of the cartesian product, it is easy to show that the two traces:*

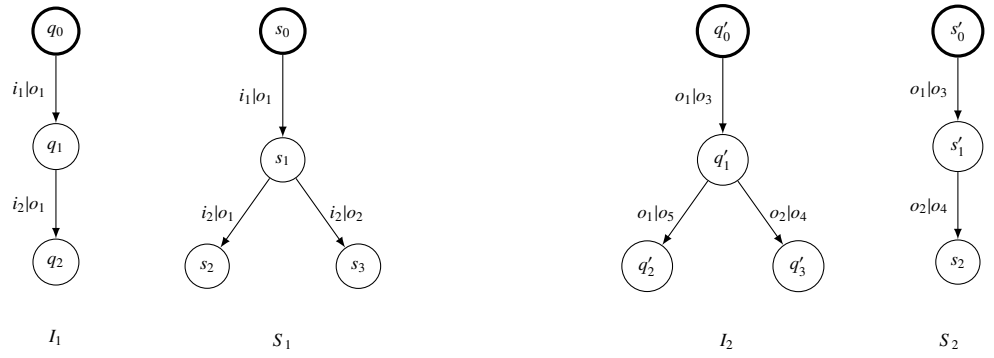
$$tr_1 = \langle i_1|o_1, \ldots, i_n|o_n\rangle \in Trace(I_1) \text{ and } tr_2 = \langle i'_1|o'_1, \ldots, i'_n|o'_n\rangle \in Trace(I_2)$$

are respectively the traces involved in I_1 *and* I_2 *to obtain tr. We also know by hypothesis that* $tr_1 \in Trace(S_1)$ *and* $tr_2 \in Trace(S_2)$.

Since $(o, o') \in Out(\otimes((I_1, I_2)) \text{ after } (tr, (i, i')))$ *and tr is composed of* tr_1 *and* tr_2, *then* $o \in Out(I_1 \text{ after } (tr_1, i))$ *and* $o' \in Out(I_2 \text{ after } (tr_2, i'))$. *Similarly,* $o \notin Out(S_1 \text{ after } (tr_1, i))$ *and* $o' \notin Out(S_2 \text{ after } (tr_2, i')$ *because* $(o, o') \notin Out(\otimes((S_1, S_2)) \text{ after } (tr, (i, i')))$ *and* tr_1 *and* tr_2 *are involved to obtain tr. Hence, there exists a trace* $tr_1 \in Trace(S_1)$, *an input i of* S_1 *and an output* $o \in \mathsf{Out}_1$ *such that* $o \in Out(I_1 \text{ after } (tr_1, i))$ *and* $o \notin Out(S_1 \text{ after } (tr_1, i))$ *(respectively there exists a trace* $tr_2 \in Trace(S_2)$, *and input i' of* S_2 *and an output* $o' \in \mathsf{Out}_2$ *such that* $o' \in Out(I_2 \text{ after } (tr_2, i'))$ *and* $o' \notin Out(S_2 \text{ after } (tr_2, i')))$. *Indeed, this means that* $\neg(I_1 \ cioco \ S_1)$ *and* $\neg(I_2 \ cioco \ S_2)$. *Hence, we have a contradiction with our hypothesis.*

End

Compositionality for feedback operators. Before proving the compositionality of *cioco* for both synchronous and relaxed feedback operators, we give an example that illustrates the assumptions required to obtain the compositionality of *cioco* with respect to the feedback operators. Figure 12 shows two implementation models I_1 and I_2 that have been tested to be **cioco**-correct according to their respective specification models S_1 and S_2. We can easily see that $(I_1 \ cioco \ S_1)$ and $(I_2 \ cioco \ S_2)$.



(a) I_1 *cioco* S_1 (b) I_2 *cioco* S_2

Figure 12: Counterexample of compositionality

Using the cartesian product and the feedback operator over the synchronous sequential interface $\mathcal{I} = (f, \pi_i, \pi_o)$ defined in Section 4.2.1, the global specification $S = \circlearrowleft_{\mathcal{I}}(\otimes(S_1, S_2))$ (resp. the global implementation $I = \circlearrowleft_{\mathcal{I}}(\otimes(I_1, I_2))$) can be obtained. We can easily see that I can do the trace $\langle i_1|o_3, i_2|o_5\rangle$. Thus, $o_5 \in Out(I \text{ after } (\langle i_1|o_3\rangle, i_2))$ whereas S can do the trace $\langle i_1|o_3\rangle$ in such a way $o_5 \notin Out(S \text{ after}(\langle i_1|o_3\rangle, i_2))$. Hence, we can see that I does not conform to S according to *cioco*.

This counterexample shows that the feedback operators may give rise to a global implementation that does not conform to its global specification, even if the local implementations conform to their local specifications. This is because the conformance relation *cioco* does not put any constraint on the traces that are not specified in the specification. It gives freedom to implementations to do what they want from the unspecified states. To cope with this problem, we assume that specifications are input-enabled like in [17]. That is to say, all states of a specification S accept all input actions of S, and for each state s of S and each input the function α is defined (α is a total function). Then, we have the following theorem for the compositionality for the relaxed feedback operator:

Theorem 8.2. *Let $H = T(\mathsf{Out} \times \_)^{\mathsf{In}}$ be a signature. Let $\mathcal{I} = (f, \pi_i, \pi_o)$ be a relaxed feedback interface. Let $C_j = (S_j, \alpha_j) \in \mathbf{Comp}(H)$ such that C_j are input-enabled for every $j = 1, 2$. Then, we have:*

$$C_1 \ cioco \ C_2 \implies \hookleftarrow_{\mathcal{I}}(C_1) \ cioco \ \hookleftarrow_{\mathcal{I}}(C_2) \quad (1)$$

$$C_1 \ cioco \ C_2 \implies \circlearrowleft_{\mathcal{I}}(C_1) \ cioco \ \circlearrowleft_{\mathcal{I}}(C_2) \quad (2)$$

Proof . *We first need to prove the following lemma:*

Lemma 5. *Consider two components C_1 and C_2, then we have:*

1. *$Trace(C_1) \subseteq Trace(C_2)$ implies $(C_1 \ cioco \ C_2)$*
2. *If C_2 is input-enabled, then $(C_1 \ cioco \ C_2)$ implies $Trace(C_1) \subseteq Trace(C_2)$.*

Proof .

1. *Let $tr = \langle i_1|o_1, i_2|o_2, \ldots, i_n|o_n\rangle$ be a finite trace of C_2, i an input of C_2 and $o \in Out(C_1 \text{ after } (tr, i))$ and let us prove that $o \in Out(C_1 \text{ after } (tr, i))$. $o \in Out(C_1 \text{ after } (tr, i))$ implies $tr' = tr.\langle i|o\rangle = \langle i_1|o_1, i_2|o_2, \ldots, i_n|o_n, i|o\rangle \in Trace(C_1)$ (see Definition 6.3). Since $Trace(C_1) \subseteq Trace(C_2)$, $tr' \in Trace(C_2)$. Thus, $o \in Out(C_2 \text{ after } (tr, i))$, and consequently, $Out(C_1 \text{ after } (tr, i)) \subseteq Out(C_2 \text{ after } (tr, i))$. The result then follows from the definition of cioco.*

2. *Let us prove this point by induction on the structure of a trace tr of C_1, let $tr = \langle i_1|o_1, i_2|o_2, \ldots, i_n|o_n\rangle \in Trace(C_1)$.*

   - ***Basic Step:*** *$tr = \langle\rangle$ is empty trace.*
     $tr = \langle\rangle \in Trace(C_2)$ trivially holds.
   - ***Induction Step:*** *Let us write tr as concatenation of two finite traces: $tr = \langle i_1|o_1, i_2|o_2, \ldots, i_{n-1}|o_{n-1}\rangle \cdot \langle i_n|o_n\rangle$.*
     $tr \in Trace(C_1)$ implies $o_n \in Out(C_1 \text{ after } (\langle i_1|o_1, \ldots, i_{n-1}|o_{n-1}\rangle, i_n))$. Since C_2 is input-enabled, i_n is inevitably an input of C_2 at any state s. By induction hypothesis, we have $\langle i_1|o_1, \ldots, i_{n-1}|o_{n-1}\rangle \in Trace(C_2)$ and $o_n \in Out(C_1 \text{ after } (\langle i_1|o_1, \ldots, i_{n-1}|o_{n-1}\rangle, i_n))$ then $o_n \in Out(C_2 \text{ after } (\langle i_1|o_1, \ldots, i_{n-1}|o_{n-1}\rangle, i_n))$ because $C_1 \ cioco \ C_2$. Thus $\langle i_1|o_1, \ldots, i_{n-1}|o_{n-1}, i_n|o_n\rangle \in Trace(C_2)$. Consequently, $Trace(C_1) \subseteq Trace(C_2)$.

End

Let us now prove the first point of Theorem 8.2. According to Lemma 5, we have to prove:

$$Trace(C_1) \subseteq Trace(C_2) \implies Trace(\hookleftarrow_{\mathcal{I}}(C_1)) \subseteq Trace(\hookleftarrow_{\mathcal{I}}(C_2))$$

For this, let us use the proof by induction on the length of a finite trace tr of $Trace(\hookleftarrow_{\mathcal{I}}(C_1))$. Let $tr = \langle i_0|o_0, \ldots, i_n|o_n\rangle$ be a finite trace of $\hookleftarrow_{\mathcal{I}}(C_1))$.

- ***Basic Step:*** *$tr = \langle\rangle$ is empty trace.*
  $tr = \langle\rangle \in Trace(\hookleftarrow_{\mathcal{I}}(C_2))$ trivially holds.

- ***Induction Step:*** *Let us write tr as the concatenation of two finite traces:* $tr = \langle i_0|o_0, i_1|o_1, \ldots, i_{n-1}|o_{n-1}\rangle \cdot \langle i_n|o_n\rangle$.

  $tr \in Trace(\hookleftarrow_I(C_1))$ *implies, according to the definition of relaxed feedback (see Definition 4.3), that there exists an input sequence x and a couple* $(\bar{x}, y_{\bar{x}})$ *inductively defined from a finite sequence of states* $(s_0, s_1, \ldots, s_n)$ *of* S_1 *as follows:*

  - $\bar{x}(0) = x(0)$ *and* $y_{\bar{x}}(0) \in \eta'_{\mathsf{Out}\times S_1}(\alpha_1(s_0)(x(0)))_{|_1}$
  - $\forall j, 0 < j \leq n,\ \bar{x}(j) = f(x(j), y_{\bar{x}}(j-1)),\ y_{\bar{x}}(j) \in \eta'_{\mathsf{Out}\times S_1}(\alpha_1(s_j)(\bar{x}(j)))_{|_1}$ *and* $s_j \in \eta'_{\mathsf{Out}_1\times S_1}(\alpha_1(s_{j-1})(\bar{x}(j-1)))_{|_2}$

  and, for every $0 \leq j \leq n,\ \pi_i(\bar{x}(j)) = i_j$ *and* $\pi_o(y_{\bar{x}}(j)) = o_j$.

  By induction hypothesis, $\langle i_0|o_0, \ldots, i_{n-1}|o_{n-1}\rangle \in Trace(\hookleftarrow_I(C_2))$ *because* $\langle i_0|o_0, \ldots, i_{n-1}|o_{n-1}\rangle \in Trace(\hookleftarrow_I(C_1))$. *Then, similarly as above, there exists an input sequence* x' *and a couple* $(\bar{x'}, y_{\bar{x'}})$ *inductively defined from a finite sequence of states* $(s'_0, s'_1, \ldots, s'_n)$ *of* S_2 *as follows:*

  - $\bar{x'}(0) = x'(0)$ *and* $y_{\bar{x'}}(0) \in \eta'_{\mathsf{Out}\times S_2}(\alpha_2(s'_0)(x'(0)))_{|_1}$
  - $\forall j, 0 < j \leq n-1,\ \bar{x'}(j) = f(x'(j), y_{\bar{x'}}(j-1)),\ y_{\bar{x'}}(j) \in \eta'_{\mathsf{Out}\times S_2}(\alpha_2(s'_j)(\bar{x'}(j)))_{|_1}$
    and $s'_j \in \eta'_{\mathsf{Out}\times S_2}(\alpha_2(s'_{j-1})(\bar{x'}(j-1)))_{|_2}$

  and, for every $0 \leq j \leq n-1,\ \pi_i(\bar{x'}(j)) = i_j$ *and* $\pi_o(y_{\bar{x'}}(j)) = o_j$.

  Since $Trace(C_1) \subseteq Trace(C_2)$, $\langle i_0, \ldots, i_n\rangle$ *is inevitably an input sequence of* C_2, $\eta'_{\mathsf{Out}\times S_2}(\alpha_2(s'_n)(f(i_n, y_{\bar{x'}}(n-1)))_{|_1}$ *is well defined.*

  Now, we know that:

$$\eta'_{\mathsf{Out}\times S_1}(\alpha_1(s_n)(f(x(n), y_{\bar{x}}(n-1))))_{|_1} \subseteq \eta'_{\mathsf{Out}\times S_2}(\alpha_2(s'_n)(f(i_n, y_{\bar{x'}}(n-1)))_{|_1}$$

  This is because $Trace(C_1) \subseteq Trace(C_2)$. *This implies that* $y_{\bar{x}}(n) \in \eta'_{\mathsf{Out}\times S_2}(\alpha_2(s'_n)(f(i_n, y_{\bar{x'}}(n-1))))_{|_1}$. *Hence according to the definition of relaxed feedback,* $\langle i_1|o_1, \ldots, i_{n-1}|o_{n-1}, i_n|o_n\rangle \in Trace(\hookleftarrow_I(C_2))$. *Consequently,* $Trace(\hookleftarrow_I(C_1)) \subseteq Trace(\hookleftarrow_I(C_2))$.

  Similarly, we can prove the second point of Theorem 8.2: C_1 *cioco* $C_2 \implies \circlearrowleft_I(C_1)$ *cioco* $\circlearrowleft_I(C_2)$.

End

Theorems 8.1 and 8.2 obviously lead to the following theorem:

Theorem 8.3. *Let op be a complex operator of arity n. Let* $C_1, \ldots, C_n, C'_1, \ldots, C'_n$ *be input-enabled components such that* $\forall i, 1 \leq i \leq n$, C_i *cioco* C'_i, *then one has* $op(C_1, \ldots, C_n)$ *cioco* $op(C'_1, \ldots, C'_n)$.

Proof . *By induction on the structure of op using Theorem 8.1 and Theorem 8.2.*
End

By Theorem 8.3, we directly have that sequential and concurrent compositions as well as synchronous product are compositional for *cioco*.

9. Conclusion and related works

9.1. Related works

In this section, we present a brief overview of contributions which are technically close to our approach, by discussing the difference between problematics addressed by those contributions and those addressed by our approach. There are several coalgebraic works in the literature which regard the combination of components using some sort of integration mechanism. The closet of our works is the set of integration operators proposed by *Barbosa* in [3, 4]. Four component integration operators have been proposed to reason about component based designs: pipeline "series" operator, external choice operator, parallel composition and concurrent operator. These operators are defined as special functors in some bicategory of components. The pipeline operator is similar to our synchronous sequential operator $\triangleright_r$. The external choice operator corresponds to a composition where both components C_1 and C_2 are executed independently, depending on the input submitted to the integrated component: when interacting with the composed system, the environment will be allowed to choose either C_1 or C_2 inputs, but not both. Input then triggers the corresponding component (i.e. C_1 or C_2), producing the associated output. This operator is then similar to our synchronous product $\circledast$ when the intersection of input sets In_1 and In_2 is the empty set. The parallel composition is embodied in the cartesian product, and finally the concurrent operator is similar to the operator defined in Section 4.2.3.

Meng in [46] redefined *Barbosa*'s operators to combine two components C_1 and C_2 over the signatures $(\mathsf{Out}_1 \times T(\mathsf{Out}_2 \times \_)^{\mathsf{In}})$ and $(\mathsf{Out}'_1 \times T'(\mathsf{Out}'_2 \times \_)^{\mathsf{In}'})$ respectively. Hence, the difference between *Meng*'s works and *Barbosa*'s ones is the form of the functor over which components are defined, and the possibility to combine components with different computational models (i.e. T and T'), rather than using a single monad.

In this paper, we have also shown how to define larger systems by composition of subsystems from two basic integration operators: product and feedback. This led us to inductively define a set of complex operators (see Definition 4.7), the semantics of which are partial functors on categories of components. This part can then be compared to works in [47, 48]. Indeed, from set of complex operators we can easily generate an algebraic signature that can be seen as an FP-theory $\mathbb{L}$ over a basic set of sorts $S \subseteq \mathbf{Set} \times \mathbf{Set}$ where for $(\mathsf{In}, \mathsf{Out}) \in S$, In and Out denote input and output sets, respectively, and operations are complex operators (a monad T is supposed identical for every couple $(\mathsf{In}, \mathsf{Out})$ in the FP-theory $\mathbb{L}$). Outer models can then be defined along the functor $\mathbb{C} : \mathbb{L} \longrightarrow \mathbf{Cat}$ that associates to any couple $(\mathsf{In}, \mathsf{Out})$ the category $\mathbf{Comp}(H)$ with $H = T(\mathsf{Out} \times \_)^{\mathsf{In}}$ and to any operator the partial functor defined in Definition 4.7. Finally, inner models are defined by the natural transformation $X : \mathbf{1} \Longrightarrow \mathbb{C}$ where $\mathbf{1}$ is the constant functor that associates to any $S \in \mathbb{L}$ the trivial object category $\mathbf{1}$, which to any couple $(\mathsf{In}, \mathsf{Out})$ associates the final object in $\mathbf{Comp}(H)$ and to any complex operator op, the mapping on behaviours noted $[[op]]$ in [47, 48] that contains op semantics on both components and transfer functions.

The difference between our works and those mentioned above is to have defined integration operations by composition of two elementary operators, product and feedback. The interest was then to demonstrate a set of general properties on these integration operators such as the results of compositionality, by showing that these properties are valid for the product and feedback and are preserved by composition.

Hence, Theorem 5.3 is similar to Theorem 4.7 in [48] at least in these goals to establish a generic result of compositionality independent of a given integration operator.

9.2. Conclusion

In this paper, we have defined a formalism based on *Barbosa*'s component definition [3, 4]. We have then defined for this formalism a trace semantic from causal functions as this is usually done in control theory and dynamic systems design. The resulting formalism is then generic enough to subsume a large family of state-based formalisms. For this formalism, a number of theoretical results were obtained. First, in order to deal with large systems, we defined the notion of integration operator as the composition of two basic operators, the product and feedback. We then showed generic results of compositionality independently of a given integration operator. We also obtained results related to the construction of a final object in the category of components. Taking advantage of our formalism genericity, we then defined both conformance and compositional testing theories, which by definition can be applied to any formalism instance of our framework.

Several research lines can be continued from the previous work. First, the proposed formalism is just an initial proposal of formalism to model complex systems. For its application in concrete cases, it has to be more experienced in the case of real size systems. We also have the ambition to give a mathematical framework for a discipline, called

systems engineering, that has been fully tried and tested in modeling of modern industrial systems, but is very little formalized. This will require as a preliminary to extend the formalism to take into account components heterogeneity (software, hardware, human) which are mainly characterized how inputs are handled to provide observable outputs (i.e. discretely or continuously). In the context of *B. Golden*'s thesis [49], we are defining a formalism abstract enough to unify, by using non-standard analysis techniques, different time treatment of components. Moreover, in systems engineering, two kinds of operators play a crucial role in defining systems:

1. Integration operators
2. Abstraction/simulation operators.

The first kind of operators has been widely discussed in this paper, but the second not at all. Both abstraction/simulation operators aim at structuring systems at many levels of description, from the most abstract one to the most concrete till realization. These operators are classically brought together in a only one which is similar to the operator of refinement classically used in software engineering [50, 51].

References

[1] B. Kanso, M. Aiguier, F. Boulanger, A. Touil, Testing of abstract components, in: A. Cavalcanti, D. Déharbe, M.-C. Gaudel, J. Woodcock (Eds.), ICTAC, Vol. 6255 of Lecture Notes in Computer Science, Springer, 2010, pp. 184–198.

[2] M. Aiguier, P. L. Gall, M. Mabrouki, Emergent properties in reactive systems, in: Proceedings of the 2008 15th Asia-Pacific Software Engineering Conference, IEEE Computer Society, Washington, DC, USA, 2008, pp. 273–280. doi:10.1109/APSEC.2008.28.
URL http://dl.acm.org/citation.cfm?id=1487740.1488110

[3] L. Barbosa, Towards a calculus of state-based software components, Journal of Universal Computer Science 9(8) (2003) 891–909.

[4] S. Meng, L. Barbosa, Components as coalgebras: the refinement dimension, Theor. Comput. Sci.(TCS) 351 (2) (2006) 276–294. doi:http://dx.doi.org/10.1016/j.tcs.2005.09.072.

[5] E. Moggi, Notions of computation and monads, Information and Computation 93 (1991) 55–92.

[6] S. Eilenberg, Automata, Languages and Machines, Vol. C, Academic Press, New York, 1978.

[7] G. H. Mealy, A method for synthesizing sequentiel circuits, Bell Systems Techn. Jour.

[8] R. Milner, A calculus of communicating systems, Springer-Verlag New York, Inc, secaucus, NG, USA.

[9] L. Frantzen, J. Tretmans, T. Willemse, A Symbolic Framework for Model-Based Testing, in: K. Havelund, M. Núñez, G. Rosu, B. Wolff (Eds.), Formal Approaches to Software Testing and Runtime Verification – FATES/RV 2006, no. 4262 in Lecture Notes in Computer Science, Springer, 2006, pp. 40–54.
URL http://www.cs.ru.nl/ lf/publications/FTW06.pdf

[10] C. Gaston, P. L. Gall, N. Rapin, A. Touil, Symbolic execution techniques for test purpose definition, in: M. Ü. Uyar, A. Y. Duale, M. A. Fecko (Eds.), TestCom, Vol. 3964 of LNCS, Springer, 2006, pp. 1–18.

[11] B. Jeannet, T. Jéron, V. Rusu, E. Zinovieva, Symbolic test selection based on approximate analysis, in: N. Halbwachs, L. Zuck (Eds.), in 11th Int. Conference on Tools and Algorithms for tthe Construction and Analysis of Systems(TACAS), Vol. 3440 of Lecture Notes in Computer Science(LNCS), Springer, 2005, pp. 349–364.

[12] H. H. Hansen, D. Costa, J. J. M. M. Rutten, Synthesis of mealy machines using derivatives, Electr. Notes Theor. Comput. Sci. (ENTCS) 164 (1) (2006) 27–45.

[13] T. J. C. Jard, TGV: theory, principles and algorithms, International Journal on Software Tools for Technology Transfer 7 (4) (2005) 297–315.

[14] J. Tretmans, Testing labeled transition systems with inputs and outputs., in: The 8th International Workshop on Protocol Test Systems, In Cavalli, A. and Budkowski, S., editors,, Ervy, France., 1995, pp. 461–476.

[15] J. Tretmans, Test generation with inputs, outputs and repetitive quiescence, Software - Concepts and Tools 17 (3) (1996) 103–120.

[16] V. Rusu, L. d. Bousquet, T. Jéron, An approach to symbolic test generation, in: IFM '00: Proceedings of the Second International Conference on Integrated Formal Methods, Springer-Verlag, London, UK, 2000, pp. 338–357.

[17] H. van der Bijl, A. Rensink, G. Tretmans, Compositional testing with ioco, in: A. Petrenko, A. Ulrich (Eds.), Formal Approaches to Software Testing (FATES), Vol. 2931 of Lecture Notes in Computer Science, Springer Verlag, Berlin, 2004, pp. 86–100.
URL http://doc.utwente.nl/66359/

[18] M. Barr, C. Wells (Eds.), Category theory for computing science, 2nd ed., Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1995.

[19] J. L. Fiadeiro, Categories for Software Engineering, SpringerVerlag, 2004.

[20] S. Mac Lane, Categories for the Working Mathematician, Vol. 5 of Graduate Texts in Mathematics, Springer Verlag, New York, Heidelberg, Berlin, 1971.

[21] J. J. M. M. Rutten, Universal coalgebra: a theory of systems, Theor. Comput. Sci. 249 (1) (2000) 3–80.
URL http://dx.doi.org/10.1016/S0304-3975(00)00056-6

[22] M. Barr, Terminal coalgebras in well-founded set theory, Theor. Comput. Sci. 114 (2) (1993) 299–315.

[23] E. Sontag, Mathematical control theory: deterministic finite dimensional systems (2nd ed.), Springer-Verlag New York, Inc., New York, NY, USA, 1998.

[24] C. A. R. Hoare, C. A. R. Hoare, Communicating sequential processes, Communications of the ACM 21 (1985) 666–677.

[25] A. Benveniste, G. Berry, The synchronous approach to reactive and real-time systems, in: Proceedings of the IEEE, 1991, pp. 1270–1282.

[26] G. Bernot, Testing against formal specifications: A theoretical view, in: TAPSOFT'91: Proc. of the Intl. Joint Conference on Theory and Practice of Software Development, Vol. 2, Springer-Verlag, London, UK, 1991, pp. 99–119.
[27] J. Tretmans, A formal approach to conformance testing, in: Proceedings of the IFIP TC6/WG6.1 Sixth International Workshop on Protocol Test systems VI, North-Holland Publishing Co., Amsterdam, The Netherlands, 1994, pp. 257–276.
[28] R. D. Nicola, M. C. B. Hennessy, Testing equivalences for processes, Theoretical Computer Science (TCS) 34 (1–2) (1984) 83–133.
[29] I. Phillips, Refusal testing, Theor. Comput. Sci. 50 (3) (1987) 241–284. doi:http://dx.doi.org/10.1016/0304-3975(87)90117-4.
[30] E. Brinksma, A theory for the derivation of tests, Proc. 8th Int. Conf. Protocol Specification, Testing, and Verification (PSTV VIII) (1988) 63–74.
[31] R. Langerak, A testing theory for LOTOS using deadlock detection, in: E. Brinksma, G. Scollo, C. A. Vissers (Eds.), Protocol Specification, Testing and Verification (PSTV), North-Holland, 1989, pp. 87–98.
[32] R. Milner, Communication and concurrency, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.
[33] L. Briones, E. Brinksma, A test generation framework for quiescent real-time systems, in: IN FATES 04, Springer-Verlag GmbH, 2004, pp. 64–78.
[34] S. Nogueira, A. Sampaio, A. Mota, Guided Test Generation from CSP Models, in: Proceedings of the 5th international colloquium on Theoretical Aspects of Computing, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 258–273. doi:10.1007/978-3-540-85762-4_18.
URL http://dx.doi.org/10.1007/978-3-540-85762-4_18
[35] A. W. Heerink, G. J. Tretmans, Refusal testing for classes of transition systems with inputs and outputs, in: T. Mizuno, N. Shiratori, T. Higashino, A. Togashi (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), Vol. 107 of IFIP Conference Proceedings, Chapman & Hall, London, 1997, pp. 23–38.
[36] M. van Osch, Hybrid input-output conformance and test generation, in: K. Havelund, M. Nez, G. Rosu, B. Wolff (Eds.), Formal Approaches to Software Testing and Runtime Verification, Vol. 4262 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2006, pp. 70–84.
[37] J.-C. Fernandez, C. Jard, T. Jéron, L. Nedelka, C. Viho, Using on-the-fly Verification Techniques for the Generation of Test Suites, Research Report RR-2987, INRIA (1996).
URL http://hal.inria.fr/inria-00073711/en/
[38] D. Lee, M. Yannakakis., Principles and methods of testing finite state machines—a survey, Proceedings of the IEEE 84 (8).
[39] A. Sampaio, S. Nogueira, A. Mota, Compositional verification of input-output conformance via csp refinement checking, in: ICFEM '09: Proceedings of the 11th International Conference on Formal Engineering Methods, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 20–48.
[40] A. Faivre, C. Gaston, P. L. Gall, Symbolic model based testing for component oriented systems, in: A. Petrenko, M. Veanes, J. Tretmans, W. Grieskamp (Eds.), TestCom/FATES, Vol. 4581 of Lecture Notes in Computer Science, Springer, 2007, pp. 90–106.
[41] N. Yevtushenko, T. Villa, R. K. Brayton, A. Petrenko, A. L. Sangiovanni-Vincentelli, Sequential synthesis by language equation solving, in: International Workshop on Logic and Synthesis.
[42] A. Petrenko, N. Yevtushenko, Solving asynchronous equations, in: Proceedings of the FIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XI) and Protocol Specification, Testing and Verification (PSTV XVIII), FORTE XI / PSTV XVIII '98, Kluwer, B.V., Deventer, The Netherlands, The Netherlands, 1998, pp. 231–247.
URL http://portal.acm.org/citation.cfm?id=646216.681847
[43] L. Briones, C. Pasareanu, D. Giannakopoulou, Work-in-progress assume-guarantee reasoning with ioco (2004).
URL http://doc.utwente.nl/59908/
[44] E. Clarke, D. Long, K. McMillan, Compositional model checking, in: Proceedings of the Fourth Annual Symposium on Logic in computer science, IEEE Press, Piscataway, NJ, USA, 1989, pp. 353–362.
URL http://dl.acm.org/citation.cfm?id=77350.77387
[45] B. Kanso, M. Aiguier, F. Boulanger, G. Gaston, Testing of component-based systems, Internal Report 2010-05-28-DI-FBO, Supélec, http://wwwdi.supelec.fr/internalreports/ (submitted) (2011).
URL http://wwwdi.supelec.fr/internalreports/
[46] S. Meng, B. K. Aichernig, A coalgebraic calculus for component based systems, In Proceedings of FACS'03, Workshop on Formal Aspects of Component Software, Satellite Workshop of the FM.
[47] I. Hasuo, B. Jacobs, A. Sokolova, The microcosm principle and concurrency in coalgebras, 2007. preprint, available from http://www.cs.ru.nl/ ichiro/papers, I. HASUO, B. JACOBS, AND A. SOKOLOVA.
[48] I. Hasuo, C. Heunen, B. Jacobs, A. Sokolova, Coalgebraic components in a many-sorted microcosm, in: Conference on Algebra and Coalgebra in Computer Science, 2009, pp. 64–80.
[49] M. Aiguier, B. Golden, D. Krob, Modeling of complex systems: A minimalist and unified semantics for heterogeneous integrated systems, Technical report, 2011. Submitted to the journal "Applied Mathematics and Computation" - Available at http://www.lix.polytechnique.fr/~golden/.
[50] J. A. Goguen, R. M. Burstall, Institutions: abstract model theory for specification and programming, J. ACM 39 (1992) 95–146. doi:http://doi.acm.org/10.1145/147508.147524.
URL http://doi.acm.org/10.1145/147508.147524
[51] C. A. R. Hoare, Proof of correctness of data representations, Acta Informatica 1 (1972) 271–281, 10.1007/BF00289507.
URL http://dx.doi.org/10.1007/BF00289507