Coalgebra
for the working software engineer

Luís Soares Barbosa*
INL - International Iberian Nanotechnology Laboratory
High-Assurance Software Lab - INESC TEC
Universidade do Minho
Braga, Portugal
lsb@di.uminho.pt

Abstract

Often referred to as 'the mathematics of dynamical, state-based systems', Coalgebra claims to provide a compositional and uniform framework to specify, analyse and reason about state and behaviour in computing. This paper addresses this claim by discussing why Coalgebra matters for the design of models and logics for computational phenomena. To a great extent, in this domain one is interested in properties that are preserved along the system's evolution, the so-called 'business rules' or system's invariants, as well as in liveness requirements, stating that e.g. some desirable outcome will be eventually produced. Both classes are examples of modal assertions, i.e. properties that are to be interpreted across a transition system capturing the system's dynamics. The relevance of modal reasoning in computing is witnessed by the fact that most university syllabi in the area include some incursion into modal logic, in particular in its temporal variants. The novelty is that, as it happens with the notions of transition, behaviour, or observational equivalence, modalities in Coalgebra acquire a shape . That is, they become parametric on whatever type of behaviour, and corresponding coinduction scheme, seems appropriate for addressing the problem at hand. In this context, the paper revisits Coalgebra from a computational perspective, focussing on three topics central to software design: how systems are *modelled*, how models are *composed*, and finally, how *properties* of their behaviours can be expressed and verified.

*Supported by the ERDF – European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation — COMPETE 2020 Programme and by National Funds through the Portuguese funding agency, FCT, within project POCI-01-0145-FEDER-030947.

1 Introduction

1.1 Coalgebra ...

To define an (inductive) data structure, as typically taught in a first undergraduate course on programming, one essentially specifies its 'assembly process'. For example, one builds a sequence in a data domain D, either by taking an empty list or by adjoining a fresh element to an existing sequence. Thus, declaring a sequence data type yields a function $\zeta : \mathbf{1} + D \times U \longrightarrow U$, where U stands for the data type being defined. The structured domain of function ζ captures a signature of *constructors* ($nil : \mathbf{1} \longrightarrow U$, $cons : D \times U \longrightarrow U$), composed additively (*i.e.* $\zeta = [nil, cons]$). The whole procedure resembles the way in which an algebraic structure is defined.

Reversing an 'assembly process' swaps structure from the domain to the codomain of the arrow, which now captures the result of a 'decomposition' or 'observation' process. In the example at hand this is performed by the familiar *head* and *tail selectors* joined together into

$$\alpha : U \longrightarrow \mathbf{1} + D \times U \tag{1}$$

where $\alpha = \underline{*} \lhd empty? \rhd \langle head, tail \rangle$ either returns a token $*$, when observing an empty sequence, or its decomposition in the top element and the remaining tail[1].

This reversal of perspective also leads to a different understanding of what U may stand for. The product $D \times U$ captures the fact that both the head and the tail of a sequence are selected (or *observed*) simultaneously. In fact, once one is no longer focused on how to construct U, but simply on what can be observed of it, finiteness is no longer required: both finite or infinite sequences can be observed through the process above. Therefore, U can be more accurately thought of as a *state space* of a machine generating a finite or infinite sequence of values of type D. Elements of U, in this example, can no longer be distinguished by construction, but should rather be identified when generating the same sequence. That is to say, when it becomes impossible to distinguish them through the observations allowed by the 'shape' structuring the codomain of α.

Function (1) is an example of a *coalgebra*. Its ingredients are: a carrier U (intuitively the state space of a machine), the *shape* of allowed observations, technically a functor $\mathcal{F}(X) = \mathbf{1} + D \times X$, and the observation *dynamics* given by function α, *i.e.* the machine itself. Formally, a $\mathcal{F}$-coalgebra is a pair $\langle U, \alpha \rangle$ consisting of an object U and a map $\alpha : U \longrightarrow \mathcal{F}\ U$. The latter maps states to structured collections of successor states. By varying $\mathcal{F}$, *i.e.* the shape of the underlying transitions, one may

[1] Notation $\underline{e}$ denotes the constant function $\lambda\ x.e$; the conditional 'if b then a else c' is written $a \lhd b \rhd c$.

capture a large class of semantic structures used to model computational phenomena as (more or less complex) transition systems. Going even further, $\mathcal{F}$ is not restricted to be an endofunctor in *Set*, the category of sets and functions. For example, as we will see later, the category of topological spaces emerges as the natural host for coalgebras modelling continuous systems. The study of the common properties of all these systems is the subject of *Universal Coalgebra*, as developed systematically by a number of authors from the pioneering work of J. J. M. M. Rutten [61].

A morphism between two $\mathcal{F}$-coalgebras, $\langle U, \alpha \rangle$ and $\langle V, \beta \rangle$, is a map h between carriers U and V which preserves the dynamics, *i.e.* such that $\beta \cdot h = \mathcal{F}\ h \cdot \alpha$. As one would expect, $\mathcal{F}$-coalgebras and their morphisms form a category $C_{\mathcal{F}}$ where both composition and identities are inherited from the host category C. Along this paper, C will always be *Set*, but in a few explicitly mentioned cases.

This sets Coalgebra as a suitable mathematical framework for the study of dynamical systems in both a *compositional* and *uniform* way. The qualifier *uniform* requires some extra explanation: coalgebraic concepts (*i.e.* models, constructions, logics, and proof principles) are parametric on, or *typed* by, the functor that characterises the underlying transition structure. The point is that, in Mathematics as in Software Engineering, going parametric allows us to focus on the abstract structure of a problem such that, on solving it, what we actually solve is a whole class of problems. The obvious limits of human reasoning make such an economy of resources the hallmark of rational thinking. And so we are back to Engineering.

This paper aims at introducing Coalgebra as a (conceptual) tool for the working software engineer. The title is borrowed from Saunders Mac Lane's famous book *Categories for the working mathematician* first published in 1971. Category theory is the study of mathematical structures focussed on the ways they interact rather than on what they pretend to be. Roughly speaking, categories deal with (typed) *arrows* and their composition, in the same sense that sets deal with *elements*, their aggregation and membership. The theory uncovers universal properties, through which whole families of arrows can be factored out in essentially unique ways, characterises constructions uniformly applicable to structures and their transformations, and unveils dual universes by simply reversing arrows.

Coalgebras are arrows in a category. Their theory brings to scene a mathematical space in which key ingredients of computational systems find their place: state, behaviour, observation, interaction. Objects, automata, state-based components, services, processes are part of our vocabulary to talk about systems which compute by reacting to contextual stimuli received along their overall computation. Typically, reactive systems rely on the cooperation of distributed, heterogeneous, often anonymous components organised into open software architectures prepared to survive in loosely-coupled environments and adapt to changing application requirements. In a

sense, the object of Software Engineering is nothing more than the (emergent) behaviour of computing systems, for which Coalgebra provides a suitable foundation. As Robin Milner put it in his Turing Award Lecture [50],

> *From being a prescription for how to do something – in Turing's terms a 'list of instructions', software becomes much more akin to a description of behaviour, not only programmed on a computer, but also occurring by hap or design inside or outside it.*

Indeed, the origins of Coalgebra, in its applications to Computer Science, may be traced back to Peter Aczel's attempt [1] to characterise bisimulation and providing a precise semantics to Milner's calculus of communicating systems.

1.2 ... for the working software engineer

This paper is not a systematic presentation of coalgebra theory, let alone a tutorial. My aim is much humbler: to make a case for Coalgebra in relation to three main topics unavoidable in any roadmap to Software Engineering – systems' *models*, *architectures*, and *properties*. Each of them will give me the opportunity to introduce a number of concepts and constructions in Coalgebra, as well as to provide a brief illustration based on current research developed by my research team.

Models. Models are pervasive in the engineering practice, and the software domain is not an exception. Irrespective of the myriad of (textual, diagrammatic, formal, etc.) notations used in practice, models should always be understood in the sense they are in *e.g.* school physics problem-solving. There, once a problem is understood, a mathematical model is built as an appropriate abstraction, on top of which one reasons about the behaviour of the system until a 'solution' is found. We will discuss how several variants of transition systems can be modelled coalgebraically. The characterisation of systems' behaviour, and the definition of suitable notions of equivalence for state-based systems will also be addressed. As an illustration, we will revisit recent results on modelling hybrid automata as coalgebras.

Architectures. Software architecture emerged as a proper discipline within Software Engineering from the need to explicitly consider, in the development of increasingly larger and more complex systems, their overall structure, organisation, and emergent behaviour. As a model, an architecture acts as an abstraction of a system that suppresses details of its constituents, except for those which affect the ways they use, are used by, relate to, or interact with other components. This topic will be

illustrated by revisiting an architectural calculus of state-based software components framed as generalised Mealy machines, in which the strict deterministic discipline is relaxed to capture more complex behavioural patterns. In particular, we will mention the interplay between the two basic modes in which software can be composed: sequentially and concurrently, *i.e.* along a temporal or a spatial dimension, respectively. This leads to particular instances of what is known in Mathematics as an *interchange law*. Again, some aspects will be instantiated for the less regular case of components which, like sensors in a network, exhibit forms of continuous evolution.

Properties. A plethora of logics is used in Software Engineering to support the specification of systems' requirements and properties, as well as to verify whether, or to what extent, they are enforced in specific implementations. Broadly speaking, the logics of dynamical systems are *modal*, *i.e.* they provide operators which qualify formulas as holding in a certain *mode*. In mediaeval Scholastics such modes represented the strength of assertion (e.g. 'necessity' or 'possibility'). In temporal reasoning they can refer to a future or past instant, or a collection thereof. Similarly, one may express epistemic states (e.g. 'as everyone knows'), deontic obligations (e.g. 'when legally entitled'), or spatial states (e.g. 'in every point of a surface'). Regarding dynamical systems as transformations of state spaces according to specific transition shapes, *i.e.* as coalgebras for particular functors, such modes refer to particular configurations of successor states as defined, or induced, by the coalgebra dynamics. Again, Coalgebra provides a *uniform* characterisation by letting functor $\mathcal{F}$ induce 'canonical' notions of modality and the corresponding logic. General questions in modal logic, such as the trade-off between expressiveness and computational tractability, or the relationship between logical equivalence and bisimilarity, can be addressed at this (appropriate) level of abstraction. We will revisit modal logic from a coalgebraic perspective and illustrate this discussion mentioning a logic to express properties of n-layered, hierarchical transition systems.

Paper structure. Models, architectures and properties are revisited, from a coalgebraic viewpoint, in the following sections. We will try to substantiate the claim that Coalgebra is the right mathematics to model and reason about state-based systems. On the other hand, we will argue that the coalgebraic approach is generic and compositional: constructions, techniques and tools apply to a large class of application areas and can be combined in a modular way. Finally, section 5 concludes with a brief discussion of current research directions and of what the future might bring for this area.

2 Models

2.1 State and behaviour

Although information technology became ubiquitous in modern life long before a solid scientific methodology, let alone formal foundations, has been put forward, the ultimate goal of a software engineering discipline is the development of methods, techniques and tools for formal – and preferably automatic – analysis and verification of computational systems. Analysis and verification are usually performed on suitable abstractions of the real systems, rather than on the systems themselves. Coalgebra provides a framework to build such abstractions, or *models*, as state-based transition systems parametric on a transition *shape*, or *type*, given by an endofunctor $\mathcal{F}$ in a host category. The choice of $\mathcal{F}$ determines not only the expressivity of the model, but also a canonical notion of *behaviour* and *observational equivalence*.

Consider, for example, an elementary model of an object whose internal state is observable through an *attribute* $\mathsf{at} : U \longrightarrow B$ and may evolve by reacting to external stimuli through a *method*[2] $\overline{\mathsf{m}} : U \longrightarrow U^A$. This defines a coalgebra

$$p \mathrel{\widehat{=}} \langle \overline{\mathsf{m}}, \mathsf{at} \rangle : U \longrightarrow U^A \times B$$

for the functor $\mathcal{F}\, X = X^A \times B$, known in the literature as a Moore machine. A bit of syntactic sugar recovers the usual transitional notation:

$$u \xrightarrow{a}_p u' \;\Leftrightarrow\; \overline{\mathsf{m}}\, u\, a = u' \quad \text{and} \quad u \downarrow_p b \;\Leftrightarrow\; \mathsf{at}\, u = b$$

The notion of a coalgebra *morphism* $h : p \longrightarrow p'$ boils down to the following commuting diagram

$$\begin{array}{ccc} U & \xrightarrow{p} & U^A \times B \\ {\scriptstyle h}\downarrow & & \downarrow{\scriptstyle h^A \times id} \\ V & \xrightarrow{p'} & V^A \times B \end{array} \qquad \text{or, avoiding exponentials,} \qquad \begin{array}{ccc} U & \xrightarrow{\mathsf{at}} & B \\ {\scriptstyle h}\downarrow & & \downarrow{\scriptstyle id} \\ V & \xrightarrow{\mathsf{at}'} & B \end{array} \qquad \begin{array}{ccc} U \times A & \xrightarrow{\mathsf{m}} & U \\ {\scriptstyle h \times id}\downarrow & & \downarrow{\scriptstyle h} \\ V \times A & \xrightarrow{\mathsf{m}'} & V \end{array}$$

because

$$\begin{array}{ll} & \langle \overline{\mathsf{m}}', \mathsf{at}' \rangle \cdot h \;=\; (h^A \times id) \cdot \langle \overline{\mathsf{m}}, \mathsf{at} \rangle \\ \Leftrightarrow & \quad \{ \text{ products} \} \\ & \langle \overline{\mathsf{m}}' \cdot h, \mathsf{at}' \cdot h \rangle \;=\; \langle h^A \cdot \overline{\mathsf{m}}, \mathsf{at} \rangle \\ \Leftrightarrow & \quad \{ \text{ structural equality} \} \end{array}$$

[2] Notation $\overline{f}$ stands for the curried version of a function f.

$$
\begin{array}{ll}
 & \overline{\mathsf{m}'} \cdot h \;=\; h^A \cdot \overline{\mathsf{m}} \;\wedge\; \mathsf{at}' \cdot h \;=\; \mathsf{at} \\
\Leftrightarrow & \quad \{ \text{ exponentials} \} \\
 & \overline{\mathsf{m}' \cdot (h \times id)} \;=\; \overline{h \cdot \mathsf{m}} \;\wedge\; \mathsf{at}' \cdot h \;=\; \mathsf{at} \\
\Leftrightarrow & \quad \{ \text{ curry is a bijection} \} \\
 & \mathsf{m}' \cdot (h \times id) \;=\; h \cdot \mathsf{m} \;\wedge\; \mathsf{at}' \cdot h \;=\; \mathsf{at}
\end{array}
$$

The behaviour of p, denoted in the sequel by $[\![p]\!]$, at a state $u \in U$, is revealed by successive observations (or experiments) triggered by the input of different sequences $s = [a_0, a_1, \ldots]$ in A^*:

$$\mathsf{at}\ u,\ \mathsf{at}\ (\overline{\mathsf{m}}\ u\, a_0),\ \mathsf{at}\ (\overline{\mathsf{m}}\ (\overline{\mathsf{m}}\ u\, a_0)\, a_1), \ldots$$

which entails the following recursive definition of $[\![p]\!]$:

$$[\![p]\!] u\, \underline{\mathsf{nil}} \;\widehat{=}\; \mathsf{at}\, u \quad \text{and} \quad [\![p]\!]\, u\, (cons\, \langle a, t\rangle) \;\widehat{=}\; [\![p]\!]\, (\mathsf{m}\, \langle u, a\rangle)\, t\ .$$

Therefore, behaviours are elements of B^{A^*}, and can be thought of as rooted trees whose branches are labelled by sequences of inputs in A and leaves by values in B. Moreover, they organise themselves into a Moore machine over B^{A^*},

$$\omega_{\mathcal{F}} \;\widehat{=}\; \langle \overline{\mathsf{m}}_\omega, \mathsf{at}_\omega \rangle : B^{A^*} \longrightarrow (B^{A^*})^A \times B\ .$$

where

$$
\begin{array}{rcll}
\mathsf{at}_\omega\, f & \widehat{=} & f\ nil & \textit{i.e.}\ \text{the value of the attribute before any input} \\
\overline{\mathsf{m}}_\omega\, f\, a & \widehat{=} & \lambda\, s\,.\ f(cons\langle a, s\rangle) & \textit{i.e.}\ \text{input determines subsequent evolution}
\end{array}
$$

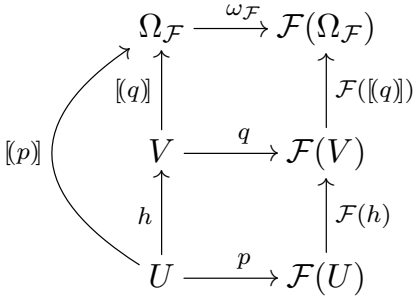
The coalgebra $\omega_{\mathcal{F}}$ whose states are the $\mathcal{F}$-behaviours themselves plays a specific role: it is *final* among all $\mathcal{F}$-coalgebras. Actually, for any $p = \langle \overline{\mathsf{m}}, \mathsf{at}\rangle$, $[\![p]\!]$ is the unique morphism $[\![p]\!] : p \longrightarrow \omega_{\mathcal{F}}$. Note that

$$
\begin{array}{ll}
 & \mathsf{at}_\omega \cdot [\![p]\!] \;=\; \mathsf{at} \\
\Leftrightarrow & \quad \{ \text{ introduction of variables } \} \\
 & \mathsf{at}_\omega \langle [\![p]\!]\, u\rangle \;=\; \mathsf{at}\, u \\
\Leftrightarrow & \quad \{ \text{ definition of } \mathsf{at}_\omega \ \} \\
 & \langle [\![p]\!]\, u\rangle\, nil \;=\; \mathsf{at}\, u \\
\Leftrightarrow & \quad \{ \text{ definition of } [\![p]\!] \ \} \\
 & true
\end{array}
\qquad\qquad
\begin{array}{ll}
 & \mathsf{m}_\omega \cdot ([\![p]\!] \times id) \;=\; [\![p]\!] \cdot \mathsf{m} \\
\Leftrightarrow & \quad \{ \text{ introduction of variables and application } \} \\
 & \mathsf{m}_\omega \langle [\![p]\!]\, u, a\rangle \;=\; [\![p]\!](\mathsf{m}\, \langle u, a\rangle) \\
\Leftrightarrow & \quad \{ \text{ definition of } \mathsf{m}_\omega \ \} \\
 & \lambda\, s\,.\ [\![p]\!]\, u\, (cons\, \langle a, s\rangle) \;=\; [\![p]\!](\mathsf{m}\, \langle u, a\rangle) \\
\Leftrightarrow & \quad \{ \text{ introduction of variables and application } \} \\
 & [\![p]\!]\, u\, (cons\, \langle a, t\rangle) \;=\; [\![p]\!]\, (\mathsf{m}\, \langle u, a\rangle)\, t \\
\Leftrightarrow & \quad \{ \text{ definition of } [\![p]\!] \ \} \\
 & true
\end{array}
$$

with uniqueness being easily established. In general, denoting by $\omega_{\mathcal{F}} : \Omega_{\mathcal{F}} \longrightarrow \mathcal{F}(\Omega_{\mathcal{F}})$ the final coalgebra for a functor $\mathcal{F}$, finality can be expressed as a universal property by the following equivalence:

$$k = [\![p]\!] \quad \Leftrightarrow \quad \omega_{\mathcal{F}} \cdot k = \mathcal{F}(k) \cdot p \tag{2}$$

Finality is a powerful tool. For example, the assertion that any two states from coalgebras p and q connected by an arbitrary morphism $h : p \longrightarrow q$ generate the same behaviour, *i.e.* $[\![p]\!] = [\![q]\!] \cdot h$, is a direct consequence of *uniqueness* (the right to left implication in equivalence (2)) as depicted in the diagram below[3].



Similarly, *existence* (the dual, left to right implication) provides a *definition* principle for operators over behaviours. Each of those has its source equipped with coalgebra structure p specifying the 'one-step' dynamics. Then $[\![p]\!]$ gives the rest: the operator becomes defined by specifying its output under all different observers as recorded in functor $\mathcal{F}$.

An important observation is that the dynamics of the final coalgebra is an isomorphism. Isomorphisms being self-dual, this also entails that the initial algebra of a functor is an isomorphism as well, which was the original statement of this result known as Lambek's lemma. The proof relies on the universal property and, in this sense, is illustrative of a proof by coinduction presented equationally.

One starts by assuming the existence of an inverse $\alpha_{\mathcal{F}}$ to $\omega_{\mathcal{F}}$, which entails $\alpha_{\mathcal{F}} \cdot \omega_{\mathcal{F}} = id_{\Omega_{\mathcal{F}}}$ and $\omega_{\mathcal{F}} \cdot \alpha_{\mathcal{F}} = id_{\mathcal{F}(\Omega_{\mathcal{F}})}$. Then, one of these requirements is used to conjecture a definition for $\alpha_{\mathcal{F}}$ (an engineer would say an 'implementation' ...). Note the use of the fact that $[\![\omega_{\mathcal{F}}]\!] = id_{\Omega_{\mathcal{F}}}$, entailing a 'reflection' law, to introduce, rather than eliminate, the behaviour morphism in the calculation. Finally, one checks the validity of the conjecture above by verifying with it the remaining requirement.

[3]The diagram captures *a fusion* property useful in behaviour reasoning.

Putting both arguments side by side, the proof goes as follows:

$$
\begin{array}{ll}
 & \alpha_{\mathcal{F}} \cdot \omega_{\mathcal{F}} = id_{\Omega_{\mathcal{F}}} \\
\Leftrightarrow & \quad \{ \text{ reflection } \} \\
 & \alpha_{\mathcal{F}} \cdot \omega_{\mathcal{F}} = [\![\omega_{\mathcal{F}}]\!] \\
\Leftrightarrow & \quad \{ \text{ universality } \} \\
 & \omega_{\mathcal{F}} \cdot \alpha_{\mathcal{F}} \cdot \omega_{\mathcal{F}} = \mathcal{F}(\alpha_{\mathcal{F}} \cdot \omega_{\mathcal{F}}) \cdot \omega_{\mathcal{F}} \\
\Leftrightarrow & \quad \{ \ \mathcal{F} \text{ preserves composition } \} \\
 & \omega_{\mathcal{F}} \cdot \alpha_{\mathcal{F}} \cdot \omega_{\mathcal{F}} = \mathcal{F}(\alpha_{\mathcal{F}}) \cdot \mathcal{F}(\omega_{\mathcal{F}}) \cdot \omega_{\mathcal{F}} \\
\Leftarrow & \quad \{ \text{ cancel } \omega_{\mathcal{F}}; \text{ universality } \} \\
 & \alpha_{\mathcal{F}} = [\![\mathcal{F}(\omega_{\mathcal{F}})]\!]
\end{array}
\qquad \Big\| \qquad
\begin{array}{ll}
 & \omega_{\mathcal{F}} \cdot \alpha_{\mathcal{F}} \\
= & \quad \{ \text{ replace by derived conjecture } \} \\
 & \omega_{\mathcal{T}} \cdot [\![\mathcal{F}(\omega_{\mathcal{F}})]\!] \\
= & \quad \{ \ [\![\mathcal{F}(\omega_{\mathcal{F}})]\!] \text{ is a morphism } \} \\
 & \mathcal{F}([\![\mathcal{F}(\omega_{\mathcal{F}})]\!]) \cdot \mathcal{F}(\omega_{\mathcal{F}}) \\
= & \quad \{ \ \mathcal{F} \text{ preserves composition } \} \\
 & \mathcal{F}([\![\mathcal{F}(\omega_{\mathcal{F}})]\!] \cdot \omega_{\mathcal{F}}) \\
= & \quad \{ \text{ just proved } \} \\
 & \mathcal{F}(id_{\Omega_{\mathcal{F}}}) \\
= & \quad \{ \ \mathcal{F} \text{ preserves identities } \} \\
 & id_{\mathcal{F}(id_{\Omega_{\mathcal{F}}})}
\end{array}
$$

Lambek's lemma characterises both initial algebras and final coalgebras for a functor $\mathcal{F}$ as fixed points of equation $X = \mathcal{F}(X)$. The terminology comes from an analogy with what happens in a partial order $\langle P, \leq \rangle$ seen as a category. A functor is then just a monotone function, and therefore a coalgebra is an element x of P such that $x \leq \mathcal{F}(x)$. The final coalgebra is, then, an element $m \leq \mathcal{F}(m)$ such that, for all $x \in P$, $x \leq \mathcal{F}(x) \Rightarrow x \leq m$, which, by Tarski's theorem, is the greatest fixpoint of $\mathcal{F}$ with respect to $\leq$.

Whenever final coalgebras exist, which is the case for every bounded *Set* endofunctor, they provide a canonical, often intuitive interpretation of behaviour. Even when this is not the case, behaviours can be approximated by an ordinal indexed sequence of objects such that each element b_α encodes behaviour that can be generated (or exhibited) in α steps.

As mentioned before, varying the functor $\mathcal{F}$ one obtains different models, with tuned notions of morphism and behaviour. For example, making $B = \mathbf{2}$ in $\mathcal{F}$ characterises deterministic automata on the alphabet A, whose behaviours are identified with the recognised languages. Actually, the state space of $\omega_{\mathcal{F}}$ becomes $\mathbf{2}^{A^*}$, *i.e.* each state is a subset of A^*, and its dynamics is given by $\langle \overline{\mathsf{m}}_\omega, \mathsf{at}_\omega \rangle : \mathbf{2}^{A^*} \longrightarrow (\mathbf{2}^{A^*})^A \times \mathbf{2}$, where

$$\mathsf{at}_\omega\, s \;=\; \mathsf{nil} \in s \quad \text{ and } \quad \overline{\mathsf{m}}_\omega\, s \;=\; \lambda\, a\,.\ \{cons\langle a, x\rangle \mid x \in s\}\ .$$

Variants of Moore machines can be obtained by specifying a particular behavioural effect $\mathcal{T}$:

$$p : U \longrightarrow \mathcal{T}(U)^A \times B$$

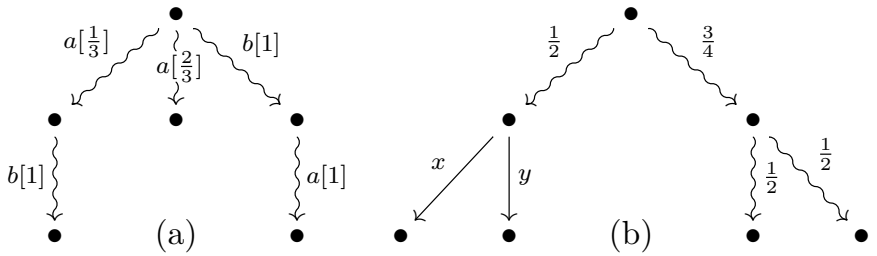
thus enforcing a particular branching structure upon p. For example $\mathcal{T}(X) = X + \mathbf{1}$

makes the automata partial, whereas $\mathcal{T} = \mathcal{P}$, for $\mathcal{P}-$ the finite, covariant powerset functor, introduces non-determinism.

Another classical distinction concerns whether the 'output' B depends on the 'input' A. For example, a generic Mealy machine would be specified by a coalgebra

$$p : U \longrightarrow \mathcal{T}(U \times B)^A \ .$$

Both Moore and Mealy machines are examples of what are usually called *reactive* transition systems, due to the explicit presence of an 'input' universe. A coalgebra for $\mathcal{F}(X) = \mathcal{T}(B \times X)$, on the other hand, stands for a so-called *generative* model, as values are produced, rather than consumed, on transitions. For example, processes in a process algebra are typically modelled as a (the final) coalgebra for $\mathcal{F}(X) = \mathcal{P}(B \times X)$. Probabilistic automata are based on the distribution functor $\mathcal{D}(X) = \{\mu : X \longrightarrow \mathbb{R}_{\geq 0} \mid \sum_{x \in X} \mu x = 1\}$. The large collection of variants of automata capturing some form of probabilistic evolution was systematically studied by Ana Sokolova [66] in a coalgebraic setting. Examples of a reactive probabilistic automata and a stratified one, in which Markovian and regular transitions may alternate, are depicted in diagrams (a) and (b) below.



The relevant functors are, respectively, $\mathcal{F}(X) = (\mathcal{D}(X) + \mathbf{1})^A$ and $\mathcal{F}(X) = \mathcal{D}(X) + (B \times S) + \mathbf{1}$. More complex transitions come from combining different effects. For example a Segala probabilistic automata is a coalgebra $p : U \longrightarrow \mathcal{P}(B \times \mathcal{D}(U))$. Note than more than one transition may be chosen non-deterministically from a given state, but once the choice is made outcomes with different probabilities are possible. Specified in a common coalgebraic setting, all such variants can be analysed and their expressivity compared through the identification of suitable natural transformations between the 'shape' functors; moreover, one typically obtains more general results and shorter proofs. Later, in subsection 2.3, recent work in my group on a similar exercise for hybrid automata will be commented. First, however, an essential ingredient for a modelling discipline is still missing: a notion of model equivalence.

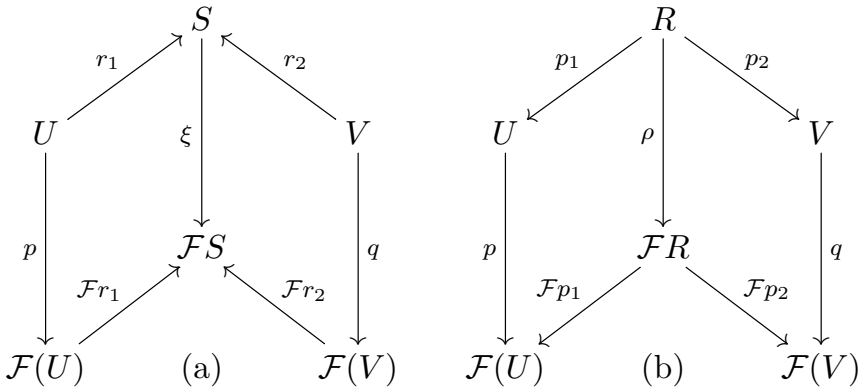
2.2 Equivalences

The comparison, replacement and reuse of models entails the need for suitable notions of equivalence. In a coalgebraic setting, this is *observational*: two states $u \in U$ and $v \in V$ in $\mathcal{F}$-coalgebras $p : U \longrightarrow \mathcal{F}(U)$ and $q : V \longrightarrow \mathcal{F}(V)$, are identified if they cannot be distinguished by observations as allowed by $\mathcal{F}$. Actually, in this case they generate the same behaviour. For example, equivalent states in a Moore machine $p : U \longrightarrow U^A \times \mathbf{2}$, *i.e.* a deterministic automaton, do recognise the same language.

Whenever $\mathcal{F}$ admits a final coalgebra $\omega_{\mathcal{F}}$, the notion of *observational equivalence*, represented in the sequel as $\equiv_{\mathcal{F}}$, can be made precise is the obvious way:

$$u \equiv_{\mathcal{F}} v \;\Leftrightarrow\; [\![p]\!]u = [\![q]\!]v \tag{3}$$

If that is not case, the definition can be generalised by requiring the existence of a coalgebra $\xi : S \longrightarrow \mathcal{F}(S)$ and a (epic) cospan $p \xrightarrow{r_1} \xi \xleftarrow{r_2} q$ in $C_{\mathcal{F}}$ (or equivalently a epic cospan in the host category C whose legs lift to $\mathcal{F}$-coalgebra morphisms, as depicted in diagram (a) below) such that $r_1 u = r_2 v$.



It is worthwhile to stress that the way $\xi : S \longrightarrow \mathcal{F}(S)$ is defined is *dual* to the one used in Algebra (the other, perhaps more familiar, half of the universe) to give a congruence. Indeed, a congruence is an equivalence relation compatible with the constructors in the algebra signature, captured by a functor $\mathcal{G}$. This means that there exists an algebra $\zeta : \mathcal{G}(A) \longrightarrow A$ and a (monic) span $a \xleftarrow{p_1} \zeta \xrightarrow{p_2} b$ to $\mathcal{G}$-algebras a and b. Not surprisingly, thus, ξ is called a *cocongruence*: congruent terms in Algebra have cocongruent behaviours as a counterpart in Coalgebra.

Interestingly enough, indistinguishability by observation is often given in terms of *bisimilarity*, *i.e.* the existence of a bisimulation containing the pair of states under consideration. A bisimulation is defined as the *analogue*, rather than the *dual*, to a compatible relation in Algebra, *i.e.* as a (monic) span $p \xleftarrow{p_1} \rho \xrightarrow{p_2} q$ in $C_{\mathcal{F}}$ as

depicted in (b). In *Set*, R is a relation in $U \times V$ whose projections p_1, p_2 lift to coalgebra morphisms, which is the original definition of bisimulation given by Aczel and Mendler [2]. We write $u \sim_{\mathcal{F}} v$ if there exists a bisimulation R such that $u = p_1 t$ and $v = p_2 t$ for a $t \in R$, and say that u and v are bisimilar states.

$$\begin{array}{ccccc} U & \xleftarrow{\ p_1\ } & G_h & \xrightarrow{\ p_2\ } & V \\ {\scriptstyle p}\downarrow & & {\scriptstyle \gamma}\downarrow & & \downarrow{\scriptstyle q} \\ \mathcal{F}(U) & \xleftarrow{\mathcal{F}(p_1)} & \mathcal{F}(G_h) & \xrightarrow{\mathcal{F}(p_2)} & \mathcal{F}(V) \end{array}$$

A rather obvious example of a bisimulation is provided by the graph of any coalgebra morphism. Indeed, let $h : p \longrightarrow q$ and $G_h = \{\langle u, hu\rangle \mid u \in U\}$, as usual. Taking, in the diagram on the right, $\gamma = \mathcal{F}(p_1)^\circ \cdot p \cdot p_1$, both squares commute because

$$\begin{array}{ll} & p \cdot p_1 = \mathcal{F}(p_1) \cdot \gamma \\ \Leftrightarrow & \quad \{ \text{ definition of } \gamma \ \} \\ & p \cdot p_1 = \mathcal{F}(p_1) \cdot \mathcal{F}(p_1)^\circ \cdot p \cdot p_1 \\ \Leftrightarrow & \quad \{ \text{ converse } \} \\ & p \cdot p_1 = p \cdot p_1 \end{array} \qquad \begin{array}{ll} & q \cdot p_2 = \mathcal{F}(p_2) \cdot \gamma \\ \Leftrightarrow & \quad \{ \text{ definition of } \gamma \ \} \\ & q \cdot p_2 = \mathcal{F}(p_2) \cdot \mathcal{F}(p_1)^\circ \cdot p \cdot p_1 \\ \Leftrightarrow & \quad \{ \ h = p_2 \cdot p_1^\circ, \text{ functors } \} \\ & q \cdot p_2 = \mathcal{F}(h) \cdot p \cdot p_1 \\ \Leftrightarrow & \quad \{ \ h \text{ is a morphism } \} \\ & q \cdot p_2 = q \cdot h \cdot p_1 \\ \Leftrightarrow & \quad \{ \ h = p_2 \cdot p_1^\circ \ \} \\ & q \cdot p_2 = q \cdot p_2 \end{array}$$

Conversely, whenever a graph G_h is a bisimulation, then h is a coalgebra morphism. Since p_1 is bijective, so is its converse p_1°. Thus, composition $h = p_2 \cdot p_1^\circ$ is a morphism.

There is an alternative definition of bisimulation which is closer to the intuitive interpretation as a binary relation over states which is closed for the coalgebra dynamics. It reads: R is a bisimulation if

$$\langle u, v\rangle \in R \ \Rightarrow \ \langle pu, qv\rangle \in \overline{\mathcal{F}}(R) \tag{4}$$

where $\overline{\mathcal{F}}(R)$ is the so-called a *relation lifting* of R through functor $\mathcal{F}$. This can be defined inductively for a wide class of functors, including all mentioned up to now in this paper, but a more general definition, applicable to any *Set* endofunctor, can be given as the image of $\mathcal{F}(R)$ under the split $\langle \mathcal{F}(p_1), \mathcal{F}(p_2)\rangle$, where p_1 and p_2 are, as before, the projections of R onto U and V, respectively; thus,

$$\overline{\mathcal{F}}(R) \ = \ \{\langle \mathcal{F}(p_1)\ t, \mathcal{F}(p_2)\ t\rangle \mid t \in \mathcal{F}(R)\} \ .$$

For example, applying (4) to the functor used above to specify Moore machines, leads to the following definition of bisimulation:

$$\langle u, v\rangle \in R \quad \Rightarrow \quad \mathsf{at}_p\ u = \mathsf{at}_q\ v \quad \text{and} \quad \langle \overline{\mathsf{m}}_p\ u\ a, \overline{\mathsf{m}}_q\ v\ a\rangle \in R, \quad \text{for all } a \in A\ .$$

This means that all states related by R support identical observations and enforce that their successor states are also related by R.

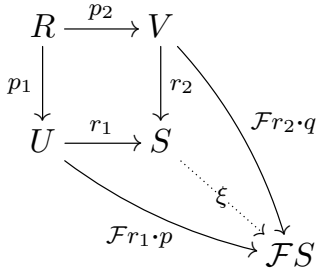
The two definitions of bisimulation discussed here are indeed equivalent. This is shown in [41] taking $\overline{\mathcal{F}}$ as a functor in a category whose objects are relations and morphisms the corresponding spans[4]. The Aczel-Mendler definition has a wider application for functors in arbitrary categories. The one based on relation lifting, on the other hand, is closer to the intuitive notion in Process Algebra [58] that a bisimulation is a closed relation.

Whatever definition one uses, the fact is that in Coalgebra bisimulation, just as behaviour, acquires a shape given by $\mathcal{F}$. Moreover, all folklore results from Process Algebra hold for coalgebraic bisimulations. In particular, the set of bisimulations linking two coalgebras forms a complete lattice for relation inclusion with joins given by unions. The largest bisimulation in this lattice is the bisimilarity relation denoted by $\sim_{\mathcal{F}}$. This is actually the greatest fixed point of a map $R \mapsto \{\langle u, v\rangle \mid \langle pu, qv\rangle \in \overline{\mathcal{F}}(R)\}$, from a direct application of the Knaster–Tarski theorem, based on $\overline{\mathcal{F}} : \mathcal{P}(U \times V) \longrightarrow \mathcal{P}(\mathcal{F}(U) \times \mathcal{F}(V))$ being monotone. Bisimulations are closed for union and converse, but not necessarily for relational composition.

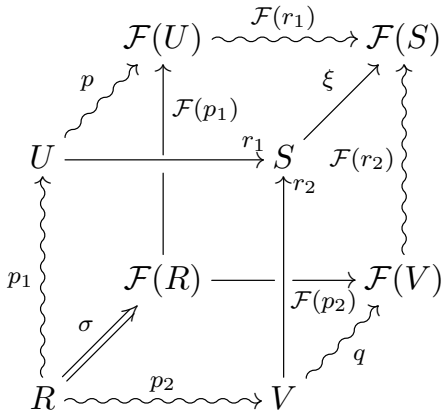
Bisimilarity, however, is strictly weaker than observation equivalence. Actually, the choice of a concept that is an analogue of, rather than a dual to, a congruence is largely motivated by historical reasons [62]. Moreover, unlike $\equiv$, bisimulations may be constructed iteratively, and therefore is amenable to automation. Quite efficient algorithms for checking bisimilarity are indeed available. For most functors of interest in current applications to Software Engineering, both notions coincide. It is instructive, however, to take a while to understand what is indeed required from the 'shape' functor in order to guarantee such a coincidence.

First of all notice it is not difficult to see that $\sim\ \subseteq \equiv$. Consider again $\mathcal{F}$-coalgebras $p : U \longrightarrow \mathcal{F}(U)$ and $q : V \longrightarrow \mathcal{F}(V)$. Now form the pushout (S, r_1, r_2) of a bisimulation R and its projections as depicted in the following diagram:

[4]Regarding $\mathcal{F}$ as a relator in the category of sets and relations leads to a very compact proof [9] of this result.



Then, arrows $\mathcal{F}(r_1) \cdot p : U \longrightarrow \mathcal{F}(S)$ and $\mathcal{F}(r_2) \cdot q : V \longrightarrow \mathcal{F}(S)$ determine a unique coalgebra ξ such that $\mathcal{F}(r_1) \cdot p = \xi \cdot r_1$ and $\mathcal{F}(r_2) \cdot q = \xi \cdot r_2$ as required. Suppose now that $u \equiv v$, *i.e.* that there is an epic cospan $p \xrightarrow{r_1} \xi \xleftarrow{r_2} q$ in $C_{\mathcal{F}}$ as depicted in the fore square of the cube in the diagram below.



Form its pullback, with $R = \{\langle u, v\rangle \in U \times V \mid r_1 u = r_2 v\}$. The lifting of this square through $\mathcal{F}$ is represented in the back square of the cube.

Coalgebras p, q and ξ link both squares. Observe that the following two paths from R to $\mathcal{F}(S)$, depicted with curly arrows in the cube, coincide:

$$
\begin{array}{ll}
 & \mathcal{F}(r_1) \cdot p \cdot p_1 \\
= & \quad \{\ r_1 : p \longrightarrow \xi \text{ is a coalgebra morphism } \} \\
 & \xi \cdot r_1 \cdot p_1 \\
= & \quad \{\ \text{the pullback square commutes } \} \\
 & \xi \cdot r_2 \cdot p_2 \\
= & \quad \{\ r_2 : q \longrightarrow \xi \text{ is a coalgebra morphism } \} \\
 & \mathcal{F}(r_2) \cdot q \cdot p_2
\end{array}
$$

If $\mathcal{F}(R)$, together with $\mathcal{F}(p_1)$ and $\mathcal{F}(p_2)$, is also a pullback, then, given the equality just proved, there exists a coalgebra $\sigma : R \longrightarrow \mathcal{F}(R)$ and coalgebra morphisms

$p_1 : \sigma \longrightarrow p$ and $p_2 : \sigma \longrightarrow q$. This means that R is a bisimulation. Notice there is no need for σ to be unique, therefore all one has to require from functor $\mathcal{F}$ is that it preserves *weak* pullbacks.

Under this apparently weird condition, bisimilarity and observational equivalence coincide. That is to say, two states generate the same behaviour if and only if they are bisimilar. Therefore, every bisimulation over the final coalgebra is a coreflexive, *i.e.* a subset of the identity relation. Furthermore this condition guarantees that the relational composition of two bisimulations is still a bisimulation, as one is used to from the (well-behaved) domain of Process Algebra.

Most *Set* endofunctors useful for the software engineer, which do indeed preserve weak pullbacks, belong to the class of extended polynomial functors

$$\mathcal{F} \ni Id \mid K \mid Id^K \mid \mathcal{P} \mid \mathcal{F} \times \mathcal{F} \mid \mathcal{F} + \mathcal{F} \mid \mathcal{F} \cdot \mathcal{F}$$

where K is a set, and $\mathcal{P}$ is the finite, covariant powerset functor. The distribution functor $\mathcal{D}$, mentioned above, is also often considered, as well as the star functor and other solutions of datatype equations.

Bisimilarity provides a technique for coinductive proofs, *i.e.* a sound tool to establish observational equivalence, which is complete for the class of functors preserving weak pullbacks. To establish equality of the behaviour generated by two state values it is enough to build a bisimulation containing them. This corresponds to the following procedure: i) iteratively strengthen the statement to be proved (from equality $u = v$ to a larger set containing the pair $\langle u, v \rangle$), and then ii) ensure that such a set is closed for the coalgebra dynamics (*i.e.* it forms a bisimulation). Actually what is going on underneath is an *unfolding* process which, typically, does not terminate, but reveals longer and longer prefixes of the result: every element in the result gets uniquely determined along this process. Inductive reasoning requires that, by repeatedly unfolding the definition, arguments become *smaller*, *i.e.* closer to the elementary constructors of the algebra. In Coalgebra our attention shifts from argument's structural shrinking to the progressive construction of the behaviour which becomes richer in informational contents.

2.3 Illustration: Hybrid automata

Hybrid automata were proposed more than two decades ago as a family of models capturing the interaction of discrete (computational) systems with continuous (physical) processes. Essentially, they are finite state machines with a finite set of continuous variables whose values are typically described by a set of ordinary differential equations. Since the publication of T. Henzinger seminal paper [36] in 1996, several different characterisations emerged independently. They were often driven

by applications, seeking to capture a specific feature or property of the system to be modelled.

As has happened before, for example in the case of probabilistic automata [66], Coalgebra helps to organise the landscape by characterising hybrid automata, and associated notions of bisimulation, in a uniform way, parametric on the concrete functor expressing the specific variant of interest. The coalgebraic perspective promotes a 'black-box' view where discrete transitions are kept internal to the automaton and continuous evolutions make up the external, observable behaviour. This is in contrast with the traditional representation in which both discrete steps and continuous evolutions are joined in the same transition relation.Therefore, the general shape for these models are coalgebras typed as

$$p : U \longrightarrow \mathcal{G}(U) \times \mathcal{H}(O) \tag{5}$$

where $\mathcal{H}$ captures the continuous evolution of a quantity O over time. Functor $\mathcal{H}$ was introduced in a recent paper [55] as an endofunctor in the category Top of topological spaces and continuous functions. In broad terms, working in Top is motivated by the key role that continuity plays in this setting, and by the possibility to handle, within the coalgebraic framework, classical properties of dynamical systems. For example, a notion of *robustness* (a system is robust if small changes in the input lead to very similar evolutions) can be addressed by varying the topology on the space of inputs. The topic, however, will not be pursued in detail here.

The functor $\mathcal{H}$ is defined as

$$\mathcal{H}(X) \;\widehat{=}\; \{\, \langle f, d\rangle \in X^{\mathsf{T}} \times \mathsf{D} \mid f \cdot \curlywedge_d = f \,\} \quad \text{and} \quad \mathcal{H}(h) \;\widehat{=}\; h^{\mathsf{T}} \times id \tag{6}$$

where T abbreviates $\mathbb{R}_{\geq 0}$, $\mathsf{D} = [0, \infty]$ is the one-point compactification of $\mathbb{R}_{\geq 0}$ and $h^{\mathsf{T}}\, f = h \cdot f$. Condition $f \cdot \curlywedge_d = f$, for $\curlywedge_d \;\widehat{=}\; id \lhd (\leq_d) \rhd \underline{d}$, means that f becomes constant after time instant d.

To illustrate this model, consider a bouncing ball dropped at some positive height and with no initial velocity. Due to the gravitational pull, it will fall into the ground but then bounce back up, losing, of course, part of its kinetic energy in the process.

This can be seen as a hybrid component whose (continuous) observable behaviour is the evolution of its spacial position (P), whereas the internal memory records the initial velocity (V) and position updated at each bounce:

$$b : V \times P \longrightarrow (V \times P) \times \mathcal{H}(P)$$

The discrete behaviour $b_d : V \times P \to V \times P$ (which updates the discrete state, *i.e.* the initial velocity and position pair) is computed by multiplying the current velocity

by the coefficient of restitution to obtain the new initial velocity for the next bounce and updating position to 0. Formally,

$$b_d \langle v, p\rangle \mathrel{\widehat{=}} \langle vel_g\langle v, zpos_g\langle v, p\rangle\rangle \times -0.5,\ 0\rangle$$

where 0.5 is the coefficient of restituion, and current velocity is computed as $vel_a\langle v, t\rangle \mathrel{\widehat{=}} v - at$. Function $zpos_a\ (v, p) \mathrel{\widehat{=}} \frac{\sqrt{2ap+v^2}+v}{a}$ returns the time needed to reach the ground, given a positive height and a current velocity. On the one hand, the continuous part $b_c : V \times P \to \mathcal{H}P$ is computed by

$$b_c \mathrel{\widehat{=}} \langle pos_g(v, p), zpos_g\rangle\ ,$$

where $pos_a : V \times P \longrightarrow P^T$ is given by $pos_a\ \langle v, p\rangle \mathrel{\widehat{=}} \lambda\, t.\ (p + vt - \frac{1}{2}at^2)$, and g is the gravitational constant. Putting both components together

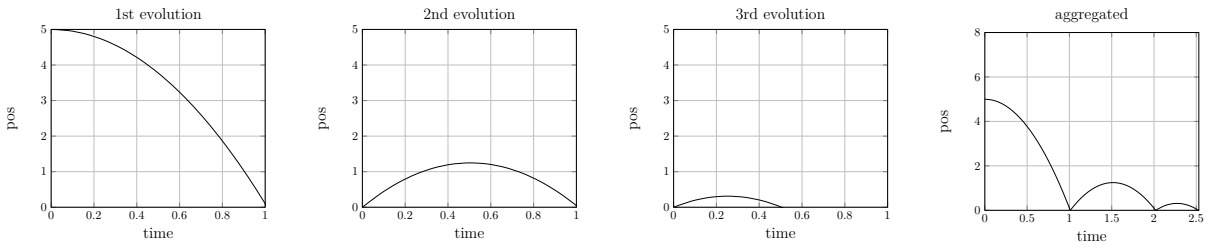
$$b \mathrel{\widehat{=}} \langle b_d, b_c\rangle\ .$$

The behaviour of a $(- \times \mathcal{H}(O))$-coalgebra p at a state $u \in U$, is a function that computes a stream of (observable) continuous evolutions generated by p from state u. Actually, the functor $- \times \mathcal{H}(O)$ has a final coalgebra, *i.e.* the following diagram commutes uniquely

$$\begin{array}{ccc} \mathcal{H}(O)^\omega & \xrightarrow{\ \omega\ } & \mathcal{H}(O)^\omega \times \mathcal{H}(O) \\ {\scriptstyle [\![p]\!]}\uparrow & & \uparrow{\scriptstyle [\![p]\!]\times id} \\ U & \xrightarrow{\ p\ } & U \times \mathcal{H}(O) \end{array}$$

where X^ω denotes the set of streams over X, and $\omega \mathrel{\widehat{=}} \langle tail, head\rangle$ is the dynamics of the final coalgebra.

For the bouncing ball, assuming the initial velocity and position pair is $\langle 0, 5\rangle$, the plot below depicts the first three elements of the generated stream.

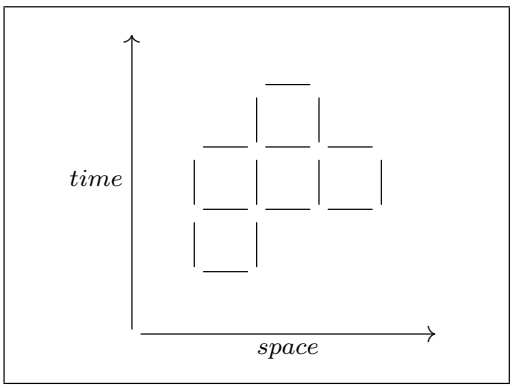


Varying $\mathcal{G}$ in (5), one is able to capture different variants of hybrid automata and compute the corresponding notions of behaviour and bisimulation. For example, instantiating $\mathcal{G}$ to $\mathcal{P}$ or $\mathcal{D}$, leads to non-deterministic or reactive Markov hybrid automata. More complex variants, already studied in the literature [28], combine non-determinism with probabilities, in the sense that, at each transition, a distribution function over states is non-deterministically chosen. They come up as coalgebras $p : U \longrightarrow \mathcal{P}\mathcal{D}(U) \times \mathcal{H}(O)$. Another interesting case makes $\mathcal{G}(U) = K^U$, for K a set of weights, thus associating costs to discrete transitions. New types of hybrid automata can also be studied in this setting. For example, $\mathcal{G} = \Delta$, for Δ the diagonal functor, gives rise to arrows of type $p : U \longrightarrow \Delta U \times \mathcal{H}(O)$, explored in [54]. These correspond to deterministic hybrid automata able to replicate themselves at each discrete transition to capture, for example, cellular replication when an organism reaches a specific saturation.

3 Architectures

3.1 Composition and refinement

Coalgebra provides a uniform framework for modelling state-based systems. The architectural problem in Software Engineering addresses the ways in which such systems can be composed. Composition has a 'geometrical' flavour: components have boundaries (*i.e.* interfaces) and organise themselves in two dimensions, *temporal* and *spatial*, as in a Cartesian plane, as depicted below. The boundary shared by vertically composed components represents handling control from one, which is terminating a particular execution thread, to another which is launching a new one. Dually, horizontal composition corresponds to concurrent evolutions being juxtaposed or eventually interacting through exchange of values or a common involvement in shared actions.



The ways in which these two basic forms of composition are defined certainly varies. For example, if components are terminating imperative programs, vertical interfaces are often realised through shared, global state variables. On the other hand, horizontal boundaries may represent the history of recorded interactions. In components modelled coalgebraically [7, 10], vertical composition is a form of pipelining (interfacing through shared output-input types), whereas horizontal composition is realised by

some form of parallel paired evolution.

At this abstract level, architectural calculi build on these two forms of composition, hereby represented by ; and $\Box$, respectively. Associativity of both operators is clearly expected, as the 'geometry' is invariant for the ways parentheses are placed. Commutativity of $\Box$ conforms to the intuition that parallel composition is unordered, but this conceptual assumption may be differently interpreted by different tensors. A similar observation applies to idempotency: it would be expected when $\Box$ encodes a choice between alternative components, but less so for a synchronous product. The interaction between the temporal and the spatial dimensions is captured by distribution, which follows the same pattern of an *interchange* law in category theory,

$$(p \,\Box\, q)\,;\,(p' \,\Box\, q') \;=\; (p\,;\,p') \,\Box\, (q\,;\,q') \;. \tag{7}$$

The law is rather familiar. For example, taking $\Box$ as parallel composition in a process algebra and ; as action prefixing, it introduces interleaving on moving from the right to the left hand side.

In practice, software architectures are described through a myriad of concrete languages and formalisms, often of a graphical nature which may allow for rather flexible coordination patterns. Coalgebra has been used to provide semantics to some of these formalisms, from classical process algebra [11] and the π-calculus [51] to statecharts [29] and different UML diagrams. In each context, composition operators to build new (coalgebraic models of) components from old, like variants of $\Box$ and ;, are specified and their properties studied. A concrete component calculus will be reviewed below to illustrate a possible application of Coalgebra to architectural design. Before that, however, we would like to address two main issues on this discussion.

The first one concerns *composition* of components modelled coalgebraically. To make things concrete, and already anticipating the illustration section, consider generalised Mealy machines

$$p : U \times I \longrightarrow \mathcal{T}(U \times O) \tag{8}$$

which can be seen as coalgebras for $\mathcal{F}(X) = \mathcal{T}(X \times O)^I$. This provides an elementary model of state-based software components characterised by

- an internal state space,
- input and output observation universes to ensure the flow of data,
- the possibility of interaction with other components during the overall computation,

- a behavioural effect which specifies their branching structure.

We have already seen some possible variants for $\mathcal{T}$. However, if one wants to compose this sort of components and define a calculus to reason about their interconnection, the first step is to enforce extra structure upon $\mathcal{T}$, namely that of a strong *monad*. Reacting to an input i, the coalgebra will not simply produce an output and a continuation state, but a $\mathcal{T}$-structure of such pairs. The monadic structure provides tools to handle the 'nesting' of such computations. Unit (η) and multiplication (μ) correspond, respectively, to a value embedding and a 'flatten' operation to reduce nested behavioural effects. The latter is the key element to define effect-aware composition, which, as discussed below, is based on composition in the Kleisli category for $\mathcal{T}$. Recall, for future reference, the definition: the composition of two monadic arrows $m : I \longrightarrow \mathcal{T}(Z)$ and $n : Z \longrightarrow \mathcal{T}(O)$, is given by $n \bullet m \mathrel{\widehat{=}} \mu \cdot \mathcal{T}(n) \cdot m$. Therefore, in this setting, the abstract operator **;** builds on composition in the universe of $\mathcal{T}$-computations, *i.e.* in the corresponding Kleisli category, whereas $\Box$ will be a tensor in this category. Strength, either in its right ($\tau_r : U \times \mathcal{T}(V) \longrightarrow \mathcal{T}(U \times V)$) or left ($\tau_l$) version, handles context information. A sort of distributive law $\delta : \mathcal{T}(U) \times \mathcal{T}(V) \longrightarrow \mathcal{T}(U \times V)$ is obtained by composing the right and left strengths. Whenever the order in which this composition is performed does not matter, the monad is said to be commutative. As one may guess this will impact on commutativity of tensors $\Box$ connecting such models.

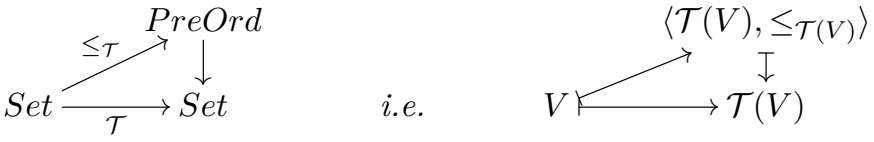
The second comment which is in order concerns the interpretation of the equality symbol in equation (7). In a coalgebraic setting the obvious choice is observational equality for the relevant functor $\mathcal{T}$. In engineering practice, however, one is sometimes interested in establishing weaker relationships. For example, (7) may be presented as an inequation to convey the intuition that a sequential computation is a special case of a parallel one, as in concurrent Kleene algebra [37].

This opens an important issue in architectural design – *refinement* – that we will briefly review from two different perspectives.

In data refinement, there is a 'recipe' to identify a refinement situation: look for an abstraction function to witness it. In other words: look for a morphism in the relevant category from the 'concrete' to the 'abstract' model such that the latter can be recovered from the former up to a suitable notion of equivalence, though typically not in a unique way. In a coalgebraic framework, however, some extra care is in order. The reason is obvious: coalgebra morphisms entail bisimilarity. Therefore one has to look for a somewhat *weaker* notion of a morphism between coalgebras.

The first approach to be mentioned here [68] works for extended polynomial endofunctors in *Set* and resorts to the notion of a preorder $\leq$ on a functor $\mathcal{T}$. This

is itself a functor which makes the following diagram commute:



This means that for any function $h : V \longrightarrow U$, $\mathcal{T}(h)$ preserves the order, *i.e.* in a pointfree formulation, $\mathcal{T}(h) \cdot \leq_{\mathcal{T}(V)} \subseteq \leq_{\mathcal{T}(U)} \cdot \mathcal{T}(h)$.

Given two $\mathcal{T}$-coalgebras $\beta : V \longrightarrow \mathcal{T}(V)$ and $\alpha : U \longrightarrow \mathcal{T}(U)$, one may now define, with respect to a preorder $\leq$, the notions of a *preserving* and a *reflecting* morphism as a function h from V to U such that

$$\mathcal{T}(h) \cdot \beta \;\dot{\leq}\; \alpha \cdot h \quad \text{and} \quad \alpha \cdot h \;\dot{\leq}\; \mathcal{T}(h) \cdot \beta \,,$$

respectively. The notation $\dot{\leq}$ is used for the pointwise lifting of the preorder $\leq$ to the functional level, *i.e.* $f \;\dot{\leq}\; g \Leftrightarrow \forall_x \,.\; f\; x \leq g\; x$,or, equivalently, $f \;\dot{\leq}\; g \Leftrightarrow f \subseteq \leq \cdot\, g$. The names chosen for these morphisms come from the fact that indeed they respectively preserve and reflect state transitions induced by coalgebras, *i.e.*

$$v \longrightarrow_\beta v' \;\Rightarrow\; h\; v \longrightarrow_\alpha h\; v' \quad \text{and} \quad h\; v' \longrightarrow_\alpha u' \Rightarrow\; \exists_{v' \in V}.\; v \longrightarrow_\beta v' \;\wedge\; u' = h\; v'$$

where $u' \longrightarrow_\alpha u \Leftrightarrow u' \in_{\mathcal{T}} \alpha\; u$ is an instance of datatype membership [38], defined inductively for the class of relevant functors [8] and verifying

$$h \cdot \in_{\mathcal{T}} \;=\; \in_{\mathcal{T}} \cdot\, \mathcal{T}\; h \tag{9}$$

for any function h.

A *refinement preorder* is a preorder $\leq$ on an endofunctor $\mathcal{T}$ satisfying the following compatibility condition with the membership relation: for all $x \in X$ and $x_1, x_2 \in \mathcal{T}(X)$,

$$x \in_{\mathcal{T}} x_1 \;\wedge\; x_1 \leq\; x_2 \;\Rightarrow\; x \in_{\mathcal{T}} x_2$$

or, again in a pointfree formulation,

$$\in_{\mathcal{T}} \cdot \leq \;\subseteq\; \in_{\mathcal{T}} \,. \tag{10}$$

It is easy to see that reflecting morphisms form a category and similarly for the dual case. The point, however, is that the exact meaning of a refinement assertion $p \leq q$ above depends on the concrete refinement preorder adopted. But what do we know about such preorders?

Condition (10) is equivalent to $\leq \subseteq \in_{\mathcal{T}} \setminus \in_{\mathcal{T}}$ by direct application of the Galois connection which defines relational *division*, *i.e.* $R \cdot X \;\subseteq\; S \Leftrightarrow X \;\subseteq\; R \backslash S$. Clearly,

this provides an upper bound for refinement preorders, the lower bound being the identity. Note that $\in_{\mathcal{T}} \setminus \in_{\mathcal{T}}$ corresponds to the lifting of $\in_{\mathcal{T}}$ to (structural) inclusion, *i.e.* $x\ (\in_{\mathcal{T}} \setminus \in_{\mathcal{T}})\ y \Leftrightarrow \forall_{e \in_{\mathcal{T}} x} .\ e \in_{\mathcal{T}} y$. Different refinement preorders have been studied [8] and will not be detailed here. In broad terms, refinement based on *preserving* morphisms generalises the usual axis of *non-determinism reduction* in a functorial way. On the other hand, *reflecting* morphisms witness a similar functorial generalisation of *definition increase*.

The second approach, developed along a series of papers by I. Hasuo [31, 35, 69, 70], plays a similar game but in a different category. It applies to coalgebras $U \to \mathcal{T}(\mathcal{F}(U))$ where $\mathcal{T}$ is a monad in Set, capturing the branching behavioural effect, $\mathcal{F}$ is a functor which determines the linear-time behaviour, and a distributive law, *i.e.* a natural transformation $\lambda : \mathcal{F}\mathcal{T} \Longrightarrow \mathcal{T}\mathcal{F}$ is assumed to hold. Thus, a $\mathcal{T}\mathcal{F}$-coalgebra in Set corresponds to a $\overline{\mathcal{F}}$-coalgebra in the Kleisli category $Kleisli(\mathcal{T})$ for monad $\mathcal{T}$. Functor $\overline{\mathcal{F}}$ is the canonical lifting of $\mathcal{F}$ to $Kleisli(\mathcal{T})$ which coincides with $\mathcal{F}$ on objects and maps an arrow $h : U \nrightarrow V$ to $\overline{\mathcal{F}}(h) = \lambda_V \cdot \mathcal{F}(h) : \mathcal{F}(U) \nrightarrow \mathcal{F}(V)$. Notice that notation $U \nrightarrow V$ stands for an arrow in $Kleisli(\mathcal{T})$, *i.e.* a Set function $U \to \mathcal{T}(V)$.

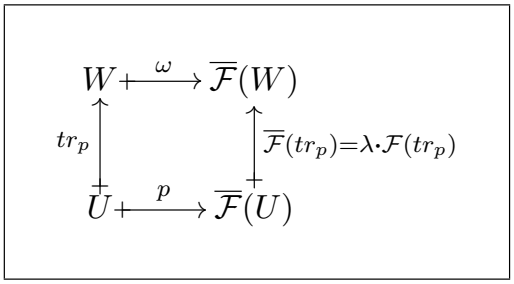
Once this setting is defined, all one has to do is to play the coalgebraic game as usual. In particular, *reflecting* and *preserving* morphisms as introduced above, emerge now as lax and oplax morphisms, renamed in this context to *forward* and *backward* simulations. However, rather than defining what we have called above refinement preorders, essentially based on the functor structure, the novelty of this approach is to build on the fact that, for the class of functors considered, the homsets in the Kleisli catagory are $\mathrm{dcpo}_{\perp}$-enriched, therefore carrying a notion of order. This means that the set of arrows, say from U to V in $Kleisli(\mathcal{T})$ forms a dcpo with a minimum element $\perp$. The crucial observation is that in such circumstances an initial $\mathcal{F}$-algebra in Set yields a final $\overline{\mathcal{F}}$-coalgebra in $Kleisli(\mathcal{T})$. Actually, this comes from Smyth and Plotkin's classical work on limit-colimit coincidence [65].

The basic result is as follows: An initial $\mathcal{F}$-algebra $\zeta : \mathcal{F}(W) \longrightarrow W$ in Set lifts to an initial $\overline{\mathcal{F}}$-algebra in $Kleisli(\mathcal{T})$, $\eta_W \cdot \zeta$, which coincides with the final $\overline{\mathcal{F}}$-coalgebra. Its dynamics is given by

$$\omega \ = \ \eta_{\mathcal{F}(W)} \cdot \zeta^{\circ} : W \nrightarrow \mathcal{F}(W)$$

in $Kleisli(\mathcal{T})$.

Coinduction in the Kleisli category works as expected, entailing a unique behaviour map from any other $\overline{\mathcal{F}}$-coalgebra $p : U \nrightarrow \overline{\mathcal{F}}(U)$ as depicted in the diagram below.



This behaviour map in the the Kleisli category, denoted by tr_p, corresponds to the *trace semantics* of the original coalgebra in Set. Just to build up intuition, let us compute tr_p for a non-deterministic automaton, *i.e.* a coalgebra $p : U \longrightarrow \mathcal{P}(\mathbf{1} + A \times U)$.

Notice that $\mathcal{T} = \mathcal{P}$ and $\mathcal{F} = \mathbf{1} + A \times -$. The carrier of the final $\overline{\mathcal{F}}$-coalgebra is $W = A^*$, as $[nil, cons] : \mathbf{1} + A \times A^* \longrightarrow A^*$ is the initial $\mathcal{F}$-algebra in Set. Therefore, $tr_p : U \longrightarrow \mathcal{P}(A^*)$ is such that

$$\begin{cases} nil \in tr_p u & \Leftarrow \ u = ß_1 * \\ cons\langle a, s\rangle \in tr_p u & \Leftarrow \ u = ß_2\langle a, u'\rangle \text{ and } s \in tr_p u' \end{cases}$$

where $ß_1, ß_2$ are the coproduct injections, which corresponds to the language accepted by the automaton p.

This example drives us in the right direction: the behaviour of a coalgebra in the Kleisli of the monad capturing the intended behavioural effect gives its *trace semantics*. Forward and backward simulations computed in exactly the same setting, as indicated above, entail notions of refinement which are sound with respect to trace inclusion (and even complete for a combination of both kinds of simulation). The point to stress, however, is that, just as the genericity of Coalgebra makes bisimulation acquire the shape of the relevant functor, it does the same to trace semantics. Similarly, different notions of simulation, *e.g.* for probabilistic and weighted coalgebras, have been extensively studied [69].

This construction of trace semantics, of which reference [35] gives a detailed account, is limited to the family of functors mentioned above. For example, it does not apply to $\mathcal{T} = \mathcal{D}$ which induces a trivial order in the Kleisli homsets; the subdistribution functor

$$\mathcal{D}_{\leq}(X) \ = \ \{\mu : X \longrightarrow \mathbb{R}_{\geq 0} \mid \sum_{x \in X} \mu x \leq 1\}$$

can be used instead – the software engineer may think of what is missing to 1, in each transition, as the probability of some sort of 'systemic' failure, such as deadlock, to occur. On the other hand, although the finite powerset monad can serve as $\mathcal{T}$ in several contexts, it cannot, for example, in combination with $\mathcal{F}(X) = A \times X$, because the initial algebra is then the empty set, thus yielding a trivial trace.

A recent, alternative path [42] to compute coalgebraic trace semantics based on *determinisation*, rather that on order enrichment, seems particularly fruitful.

The idea is borrowed from automata theory where determinisation refers to the algorithmic construction of the deterministic equivalent to a non-deterministic automata. The latter often provides a smaller representation of the problem at hand but its processing is computationally harder. A similar process converts a partial into a total automaton. Both of them have a common shape: more transitions are added but the behaviour of the non-deterministic or the partial automata is given in terms of the deterministic, total case. The coalgebraic generalization is based on a different decomposition, studying coalgebras of type $U \to \mathcal{F}(\mathcal{T}(U))$, rather than $U \to \mathcal{T}(\mathcal{F}(U))$. It puts new conditions on the functors of interest and lifts the constructions not to the Kleisli, but to the Eilenberg-Moore category of the behavioural effect monad. Interestingly enough, it captures cases that fail to have dcpo$_\perp$-enriched Kleisli homsets . One such example concerns coalgebras

$$p : U \longrightarrow \mathcal{M}(\mathbf{1} + A \times U)$$

where $\mathcal{M}(X) = \mathbb{N}^X$ is the multiset monad, and corresponds to a quite general form of weighted transition systems.

3.2 Illustration: Variants of a component calculus

My first contact with Coalgebra, in the context of my own doctoral studies, focused on the development of a calculus for software components modelled as monadic Mealy machines [7], typed as (8) above. A component model in such a setting is a pointed coalgebra

$$p \mathrel{\widehat{=}} \langle u_p \in U_p, \overline{a}_p : U_p \longrightarrow \mathcal{T}(U_p \times O)^I \rangle \tag{11}$$

where u_p is the initial state and the coalgebra dynamics is captured by currying a state-transition function $a_p : U_p \times I \longrightarrow \mathcal{T}\ (U_p \times O)$.

The basic architectural operator is *pipeline* – a form of sequential composition which amounts to the Kleisli composition for monad $\mathcal{T}$ of its arguments suitably extended to each other's state space. It is worthwhile to detail the construction. Given $a_p : U_p \times I \longrightarrow \mathcal{T}(U_p \times O)$, its (left) state extension to X is computed as

$$a_{X|p} \mathrel{\widehat{=}} (X \times U_p) \times I \xrightarrow{a^\circ} X \times (U_p \times I) \xrightarrow{id \times a_p} X \times \mathcal{T}(U_p \times O)$$
$$\xrightarrow{\tau_r} \mathcal{T}(X \times (U_p \times O)) \xrightarrow{\mathcal{T}(a)} \mathcal{T}((X \times U_p) \times O)$$

where a is the associativity natural isomorphism and τ_r the right strength for $\mathcal{T}$. The right state extension, $p|X$, is defined similarly. Therefore, for components with dynamics $a_p : U_p \times I \longrightarrow \mathcal{T}(U_p \times Z)$ and $a_q : U_p \times Z \longrightarrow \mathcal{T}(U_p \times O)$, their pipeline is defined as

$$p \,;\, q \mathrel{\widehat{=}} \langle \langle u_p, u_q \rangle, \overline{a}_{p;q} \rangle \quad \text{with} \quad a_{p;q} \mathrel{\widehat{=}} a_{U_p|q} \bullet a_{p|U_q} \ . \tag{12}$$

Having defined generic components as (pointed) coalgebras, one may wonder how do they get composed and what kind of calculus emerges from this framework. Actually, interfaces are sets representing the input and output range of a component. Consequently, components are arrows between interfaces and arrows between components are arrows between arrows. Formally, this leads to the notion of a *bicategory*[5] to structure our reasoning universe. We take interfaces (*i.e.* sets modelling observation universes of components) as *objects* of a bicategory Cp, whose *arrows* are pointed coalgebras. For each pair $\langle I, O\rangle$ of interface objects, a (hom-)category $Cp(I, O)$ is defined, whose arrows $h : \langle u_p, \overline{a}_p\rangle \longrightarrow \langle u_q, \overline{a}_q\rangle$ satisfy the expected *morphism* and *initial state preservation* conditions:

$$\overline{a}_q \cdot h \;=\; \mathcal{T}(h \times O)^I \cdot \overline{a}_p \quad \text{and} \quad h\,(u_p) \;=\; u_q\;. \tag{13}$$

Composition is inherited from Set and the identity $1_p : p \longrightarrow p$ on component p is defined as the identity id_{U_p} on its carrier. Next, for each triple of objects (I, K, O), a composition law is given by a functor

$$;_{I,K,O} : Cp(I, K) \times Cp(K, O) \longrightarrow Cp(I, O)$$

whose action on objects p and q was given above.
The action of **;** on 2-cells reduces to $h \,;\, k = h \times k$. Finally, for each object K, an identity law is given by a functor

$$copy_K : \mathbf{1} \longrightarrow Cp(K, K)$$

whose action on objects is the constant component

$$\langle * \in \mathbf{1}, \overline{a}_{copy_K}\rangle$$

with $a_{copy_K} \mathrel{\widehat{=}} \eta_{\mathbf{1}\times K}$. Similarly, the action on morphisms is the identity on $\mathbf{1}$.

The fact that, for each strong monad $\mathcal{T}$, components form a bicategory amounts not only to a standard definition of the two basic combinators **;** and $copy_K$ of a component calculus, but also to setting up its basic laws. Recall that the graph of a morphism is a bisimulation. Therefore, the existence of an initial state-preserving

[5]Basically a *bicategory* [13] is a category in which a notion of arrows between arrows is additionally considered. This means that the the space of morphisms between any given pair of objects, usually referred to as a (hom-)set, acquires itself the structure of a category. Therefore the standard arrow composition and unit laws become functorial, since they transform both objects and arrows of each hom-set in a uniform way. A typical example is Cat itself: the category whose objects are small categories, arrows are functors and arrows between arrows, or 2-cells as they are often called, correspond to natural transformations.

morphism between two components makes them bisimilar, leading to the following laws, for appropriately typed components p, q and r:

$$copy_I \,;\, p \;\sim\; p \;\sim\; p \,;\, copy_O \quad \text{and} \quad (p \,;\, q) \,;\, r \;\sim\; p \,;\, (q \,;\, r)$$

The dynamics of a component specification is essentially 'one step': it describes immediate reactions to possible state/input configurations. Its temporal extension becomes the component's *behaviour*. Formally, the behaviour $[\![p]\!]$ of a component p is computed by *coinductive extension*, *i.e.* $[\![p]\!] = [\![\overline{a}_p]\!]\ u_p$. Behaviours organise themselves in a category Bh, whose objects are sets and arrows $b : I \longrightarrow O$ are elements of the carrier of the final coalgebra $\omega_{I,O}$ for functor $\mathcal{T}(\mathsf{Id} \times O)^I$. Thus, composition in Bh is given by a family of combinators, for each I, K and O, ${;_{I,K,O}}^{Bh} : Bh(I,K) \times Bh(K,O) \longrightarrow Bh(I,O)$, such that ${;_{I,K,O}}^{Bh} \mathrel{\widehat{=}} [\![\omega_{I,K} ; \omega_{K,O}]\!]$. On the other hand, identities are given by ${copy_K}^{Bh} : \mathbf{1} \longrightarrow Bh(K,K)$ and ${copy_K}^{Bh} \mathrel{\widehat{=}} [\![\overline{a}_{copy_K}]\!]\ *$, *i.e.* the behaviour of component $copy_K$, for each K.

The basic observation is that the structure of Bh mirrors whatever structure Cp possesses. In fact, the former is isomorphic to a sub-(bi)category of the latter, whose arrows are components defined over the corresponding final coalgebra. Alternatively, we may think of Bh as constructed by quotienting Cp by the greatest bisimulation. However, as final coalgebras are fully abstract with respect to bisimulation, the bicategorical structure collapses. Moreover, as discussed in [7], some tensors in Cp become universal constructions in Bh, for particular instances of $\mathcal{T}$. This also explains why properties holding in Cp up to bisimulation, do hold 'on the nose' in the behaviour category. For example, the ; laws above may be rephrased as

$$copy_I \,;\, b \;=\; b \;=\; b \,;\, copy_O \quad \text{and} \quad (b \,;\, c) \,;\, d \;=\; b \,;\, (c \,;\, d)$$

for suitably typed behaviours b, c and d, in Bh. It is easy to check that Bh is a category and $[\![\ ]\!]$ is a 2-functor from Cp to Bh. Indeed,

$$\begin{aligned} b \,;\, copy_O &= [\![\omega_{I,O} \,;\, copy_O]\!]\ \langle b, * \rangle = [\![\omega_{I,O}]\!]\ b = b \\ (b \,;\, c) \,;\, d &= [\![(\omega_{I,K} \,;\, \omega_{K,L}) \,;\, \omega_{L,O}]\!]\ \langle \langle b, c \rangle, d \rangle = \\ &= [\![\omega_{I,K} \,;\, (\omega_{K,L} \,;\, \omega_{L,O})]\!]\ \langle b, \langle c, d \rangle \rangle = b \,;\, (c \,;\, d) \end{aligned}$$

On the other hand, note that $[\![{copy_K}^{Cp}]\!] = {copy_K}^{Bh}$ and

$$\begin{aligned} [\![(p \,;^{Cp}\, q)]\!] &= [\![\overline{a}_{p;q}]\!](\langle u_p, u_q \rangle) \\ &= [\![\omega_{I,K} \,;\, \omega_{K,O}]\!] \cdot ([\![\overline{a}_p]\!] \times [\![\overline{a}_q]\!])\ \langle u_p, u_q \rangle \\ &= {;^{Bh}} \cdot ([\![\overline{a}_p]\!] \times [\![\overline{a}_q]\!])\ \langle u_p, u_q \rangle \\ &= {;^{Bh}}\ \langle [\![\overline{a}_p]\!]\ u_p, [\![\overline{a}_q]\!]\ u_q \rangle \\ &= [\![p]\!]\ {;^{Bh}}\ [\![q]\!] \end{aligned}$$

Some detail put above in describing the structure of Cp and Bh aims at emphasising an important aspect from the architectural point of view: behaviour descriptions are compositional in a sense that is compatible with composition at the (state based) component level. Such compatibility comes exactly from the bicategorial structure. Or, as put by I. Hasuo, C. Heunen, B. Jacobs and A. Sokolova [33], as a manifestation of the *microcosm* principle which states that the same algebraic structure is carried by a category and by one of its objects which assumes a prototypical role. Examples abound in the literature on 'categorification' [4], a typical one is that of a monoid object inside a monoidal category. It is interesting that the same kind of phenomena arises in our context.

A whole component calculus, parametric on a behaviour monad $\mathcal{T}$, can be developed on Cp. The relevant structure lifts naturally to Bh defining a particular (typed) 'process' algebra. We will not go into detail here, but to mention the basic ingredients considered in all approaches documented in the literature [7, 10, 33, 32].

The first one is the representation of functions in Cp: A function $f : A \longrightarrow B$ is lifted to a component $\ulcorner f \urcorner \mathrel{\widehat{=}} \langle * \in \mathbf{1}, \overline{a}_{\ulcorner f \urcorner} \rangle$ over $\mathbf{1}$ whose action is given by the currying of

$$a_{\ulcorner f \urcorner} \mathrel{\widehat{=}} \mathbf{1} \times A \xrightarrow{id \times f} \mathbf{1} \times B \xrightarrow{\eta_{(\mathbf{1} \times B)}} \mathcal{T}(\mathbf{1} \times B)$$

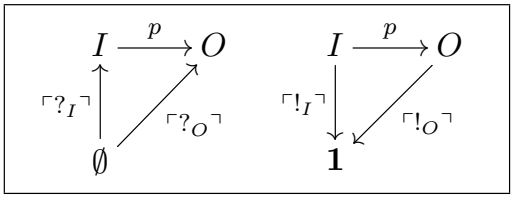
Up to bisimulation, function lifting is functorial, that is, for $g : I \longrightarrow K$ and $f : K \longrightarrow O$ functions, one has

$$\ulcorner f \cdot g \urcorner \sim \ulcorner g \urcorner ; \ulcorner f \urcorner \quad \text{and} \quad \ulcorner id_I \urcorner \sim copy_I$$

Actually, lifting canonical Set arrows to Cp is a simple way to explore the structure of Cp itself. For instance, $?_I : \emptyset \longrightarrow I$ keeps its naturality as, for any $p : I \longrightarrow O$, the corresponding diagram below commutes up to bisimulation, because both $\ulcorner ?_I \urcorner$ and $\ulcorner ?_O \urcorner$ are the *inert* components: the absence of input makes reaction impossible. Formally,

$$\ulcorner ?_I \urcorner ; p \sim \ulcorner ?_O \urcorner \tag{14}$$

Naturality is lost, however, in the lifting of $!_I : I \longrightarrow \mathbf{1}$: the diagram fails to commute for non trivial $\mathcal{T}$ (*e.g.* the finite powerset monad).



Components over $\mathbf{1}$ defined from identities and structural properties of the underlying category are called *wires*. Typical examples, include the liftings of canonical isomorphisms – *e.g.* associativity, a, or commutativity, s – which leads to bisimilarity up to an isomorphic rearranging of the interface, as well as liftings of embeddings, projections,

codiagonals and diagonals, the latter used to *merge* input and *replicate* output types, as in, for example, $\ulcorner \nabla \urcorner ; p ; \ulcorner \triangle \urcorner : I + I \longrightarrow O \times O$.

The pre- and post-composition of a component with Cp-lifted functions can be encapsulated into a unique combinator, called *wrapping*, which is reminiscent of the *renaming* connective found in process calculi. It is defined as functor $-[f, g] : Cp(I, O) \longrightarrow Cp(I', O')$, for f and g suitably typed, which is the identity on morphisms and maps component $\langle u_p, \overline{a}_p \rangle$ into $\langle u_p, \overline{a}_{p[f,g]} \rangle$, where

$$a_{p[f,g]} \mathrel{\widehat{=}} U_p \times I' \xrightarrow{id \times f} U_p \times I \xrightarrow{a_p} \mathcal{T}(U_p \times O) \xrightarrow{\mathcal{T}(id \times g)} \mathcal{T}(U_p \times O')$$

Typical properties are, as one could expect,

$$p[f, g] \sim \ulcorner f \urcorner ; p ; \ulcorner g \urcorner \quad \text{and} \quad (p[f, g])[f', g'] \sim p[f \cdot f', g' \cdot g]$$

Components can be aggregated in a number of ways, besides the 'pipeline' composition discussed above. Several tensors have been introduced in the literature [7, 33, 52] corresponding to *choice*, *parallel* and *concurrent* composition. We will briefly detail the first one which provides a form of additive composition defined as a lax functor $\boxplus : Cp \times Cp \longrightarrow Cp$. It consists of an action on objects given by $I \boxplus J = I + J$ and a family of functors $\boxplus_{I,O,J,R} : Cp(I, O) \times Cp(J, R) \longrightarrow Cp(I + J, O + R)$ yielding

$$p \boxplus q \mathrel{\widehat{=}} \langle \langle u_p, u_q \rangle \in U_p \times U_q, \overline{a}_{p \boxplus q} \rangle \quad \text{and} \quad a_{p \boxplus q} \mathrel{\widehat{=}} dr^{\circ} \bullet (a_{p|U_q} + a_{U_p|q}) \bullet dr^{\circ} ,$$

where dr is the right distributivity isomorphism, and mapping pairs of arrows $\langle h_1, h_2 \rangle$ into $h_1 \times h_2$. When interacting with $p \boxplus q$, the environment chooses either to input a value of type I or one of type J, triggering the corresponding component, p or q, respectively. The following laws arise from the fact that $\boxplus$ is a lax functor in Cp:

$$\begin{aligned} (p \boxplus p') ; (q \boxplus q') &\sim (p ; q) \boxplus (p' ; q') \\ copy_{K \boxplus K'} &\sim copy_K \boxplus copy_{K'} \\ \ulcorner f \urcorner \boxplus \ulcorner g \urcorner &\sim \ulcorner f + g \urcorner . \end{aligned}$$

Moreover, up to isomorphic wiring, $\boxplus$ is a symmetric tensor product in each hom-category, with $nil = \ulcorner id_{\emptyset} \urcorner$ as unit, *i.e.*

$$\begin{aligned} (p \boxplus q) \boxplus r &\sim p \boxplus (q \boxplus r) \\ nil \boxplus p \ &\sim \ p \ \text{ and } \ p \boxplus nil \ \sim \ p \\ p \boxplus q &\sim q \boxplus p . \end{aligned}$$

The construction of architectural calculi based on generalised Mealy machines initiated in [6], was furhter developed by a several authors. V. Miraldo and J. N. Oliveira [52] focused on lifting the whole calculus to the Kleisli category of the relevant behaviour monad $\mathcal{T}$, under the *motto* 'keep definition, change category'. Some of these categories are paradigmatic universes for dealing, namely, with non-determinism and probabilistic evolution. In the first case the calculus is 'instantiated' in the category of sets and binary relations, *i.e.* the Kleisli for the finite powerset functor. In the second, in the category of (sub-)stochastic matrices, the Kleisli of the (sub-)distribution functor. In both cases, calculation takes advantage of a well-studied universe of *typed* relations and matrices, respectively [57]. The programme is not straightforward, namely in what concerns the lifting of theories (*e.g.* of behavioural equivalence), further than just the definition of combinators and the preservation of monadic strength on moving from the original to the Kleisli category. Again, a somehow heavy requirement is found here: monad $\mathcal{T}$ should induce a $\mathrm{dcpo}_{\perp}$-enriched Kleisli category (as pointed out above when discussing trace semantics for coalgebras) and should itself be symmetric monoidal, *i.e.* commutative.

These two requirements appear again in the approach developed by B. Jacobs, I. Hasuo, C. Heunen and A. Sokolova, in a series of papers [33, 32, 34]. The whole work is done in the context of a symmetric monoidal category C, equipped with coproducts $+$ and $\emptyset$ over which the tensor $\otimes$ distributes. The behaviour monad $\mathcal{T}$ is assumed to be commutative, with a distributive law $\delta : \mathcal{T}(U) \times \mathcal{T}(V) \longrightarrow \mathcal{T}(U \times V)$. Instead of building on a bicategorial structure as before, components are taken as objects in a category with fixed input/output universes. This entails the need for the introduction of an *indexing* mechanism, similar to the one underlying relabelling in process algebra. Actually, the calculus works directly with arrows $U \times I \to \mathcal{T}(U \times O)$ which lift to coalgebras if C is Cartesian closed. This is not assumed in general, leading to a very general setting; for example state extension discussed above arises simply as an action of the monoidal category on a category of components. A very interesting connection links components in C to Freyd categories [60], which further correspond to J. Hughes' notion of an *arrow* [39], a construction which, like that of a monad, is used to model structured computations in functional programming.

But what is really new in this approach is the introduction of a *trace* operator in the architectural calculus, which provides a formalisation of the notion of a 'loop' in a diagram of components. Semantically, this brings to scene a feedback construction with respect to the additive structure of C, embodying a form of *iteration*. Mathematically, the operator is a trace in the sense of A. Joyal, R. Street and D. Verity [43].

The development of these ideas can be summed up as follows: whenever C has countable coproducts and the behaviour monad $\mathcal{T}$ is commutative and, as before,

induces a Kleisli category whose homsets are dcpo$_\perp$ enriched, $Kleisli(\mathcal{T})$ is traced monoidal with respect to coproduct as a monoidal structure. This means that, to each arrow $h : I+Z \longrightarrow \mathcal{T}(O+Z)$ corresponds a traced arrow $Tr^{Kl}(h) : I \longrightarrow \mathcal{T}(O)$, where operator $Tr^{Kl}(\ )$ satisfies the canonical properties for a trace [43]. This lifts to a trace operator in the category of components which also obeys those properties, although only up to isomorphism. Thus, given a component $p = \langle u_p \in U_p, \overline{a}_p \rangle : I + Z \longrightarrow O + Z$ the trace operator builds a new one where output in Z is fed back to p:

$$Tr(p) \mathrel{\widehat{=}} \langle u_p \in U_p, \overline{a}_{Tr(p)} \rangle : I \longrightarrow O \quad \text{with} \quad a_{Tr(p)} \mathrel{\widehat{=}} Tr^{Kl}(\mathcal{T}(dr^\circ) \bullet a_p \bullet dr)$$

Note that $\mathcal{T}(dr^\circ) \bullet a_p \bullet dr$ is typed as $U_p \times I + U_p \times Z \to \mathcal{T}(U_p \times O + U_p \times Z)$ in $Kleisli(\mathcal{T})$.

The requirements on the behaviour monad mentioned above seem to be recurrent when aiming at a richer structure for the development of architectural calculi. They are not met, however, by the functor $\mathcal{H}$ introduced in section 2.3 and intended to capture continuous behaviour.

The functor $\mathcal{H}$, however, extends to a strong monad (both in Set and Top) which means that most of the calculus discussed above can be developed to address components with continuous evolutions both in their output and state space, *i.e.* built as coalgebras for $\mathcal{F}(X) = \mathcal{H}(U \times O)^I$. Basically, all the calculus is kept, but for an additive trace and the interchange law with respect to a tensor capturing parallel evolution. There is, however, a notion of iteration, which, under some circumstances [55], induces a fixed point to give semantics to infinite loops.

In any case, it seems that $\mathcal{H}$-based coalgebras will play a relevant role in generalising the component calculus to the continuous domain, to reason *e.g.* about sensor networks and IoT configurations. Therefore, I'll close this section introducing the associated monadic structure. Recall the definition of $\mathcal{H}$ in (6), section 2.3. The monad structure adds a multiplication, $\mu : \mathcal{H} \cdot \mathcal{H} \Longrightarrow \mathcal{H}$ and its unit $\eta : Id \Longrightarrow \mathcal{H}$. The latter produces trivial evolutions with duration 0. Formally, $\eta_X\ x \mathrel{\widehat{=}} \langle \underline{x},\ 0 \rangle$. Multiplication is a bit more complex. Let $\langle f, d \rangle \in (X^T \times D)^T \times D$. Then, the 'flattened' system, $\mu_X\ \langle f, d \rangle$, will return, at each instant t_i, the value $(f\ t_i)0$ until, and if, d is reachead. After that, if $d \neq \infty$, it will evolve according to $fd = \langle g, e \rangle$ for the remaining duration $e - d$. Formally,

$$\mu_X\ \langle f, d \rangle \mathrel{\widehat{=}} \begin{cases} \langle \theta_X \cdot f, d \rangle \ +\!\!+\ (f\ d) & \text{if } d \neq \infty \\ \langle \theta_X \cdot f,\ \infty \rangle & \text{otherwise} \end{cases}$$

where $\theta : \mathcal{H} \Longrightarrow Id$ is given by $\theta\ \langle f, d \rangle \mathrel{\widehat{=}} f\ 0$ and $\langle f, d \rangle \ +\!\!+\ \langle g, e \rangle \mathrel{\widehat{=}} \langle f \ +\!\!+_d\ g, d + e \rangle$ with $f \ +\!\!+_d\ g \mathrel{\widehat{=}} f \lhd (\leq_d) \rhd g\ (\_ - d)$. Note that θ is an Eilenberg-Moore $\mathcal{H}$-algebra: indeed, $\theta \cdot \eta = id$ and $\theta \cdot \mu = \theta \cdot \mathcal{H}\theta$.

It is worthwhile to see what composition means in $Kleisli(\mathcal{H})$. Let $c_1 : I \nrightarrow K$, $c_2 : K \nrightarrow O$, and assume $c_i = \langle f_i, d_i \rangle$, for $i = 1, 2$. Thus[6],

$$
\begin{array}{ll}
 & (\mu \cdot \mathcal{H}c_2 \cdot c_1)\ x \\
= & \qquad \{\ c_1 = \langle f_1, d_1 \rangle, \text{ and let } d = d_1\ x\ \} \\
 & (\mu \cdot \mathcal{H}c_2)\ \langle f_1\ x, d \rangle \\
= & \qquad \{\ \text{definition of } \mathcal{H}\ \} \\
 & \mu\ \langle c_2 \cdot (f_1\ x), d \rangle \\
= & \qquad \{\ \text{definition of } \mu \} \\
 & \langle \theta \cdot c_2 \cdot (f_1\ x), d \rangle\ \ +\!\!+\ (c_2 \cdot (f_1\ x))\ d \\
= & \qquad \{\ \text{definition of } +\!\!+\ \} \\
 & \langle\ (\theta \cdot c_2 \cdot (f_1\ x))\ \ +\!\!+_d\ f_2\ ((f_1\ x)\ d),\ \ d + \pi_2(c_2((f_1\ x)\ d))\ \rangle \\
= & \qquad \{\ \text{definition of } +\!\!+_d\ \} \\
 & \langle\ \theta \cdot c_2\ ((f_1\ x)\ \_)\ \lhd\ (\leq_d)\ \rhd\ f_2\ ((f_1\ x)\ d)\ (\_ - d),\ \ d + \pi_2(c_2((f_1\ x)\ d))\ \rangle
\end{array}
$$

Two different cases must be considered. In the first one suppose that c_2 in $(c_2 \bullet c_1)$ is *pre-dynamical* in the standard sense that $\theta \cdot c_2 = id$ (or, at least, an inclusion). In this case composition yields *sequencing*: for the duration of $c_1 x$, $(c_2 \bullet c_1)x$ evolves first according to c_1, and then, on its termination, according to c_2, which receives as input the endpoint of $f_1 x$. Otherwise it yields a form of *modulation*: the second component acts upon the first one.

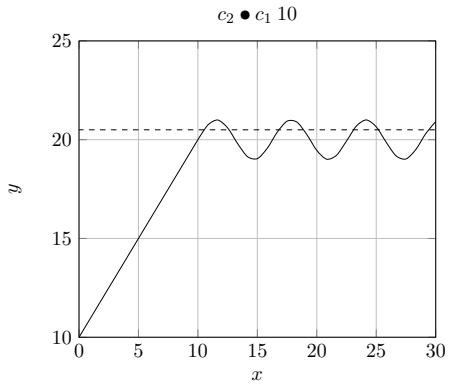
This behaviour may be illustrated through the following example. Suppose the temperature of a room is to be regulated as follows: start at 10 °C, seek to reach and maintain 20 °C, but in no case surpass 20.5 °C. The system is realised by three elementary components that have to work together: component c_1 to raise the temperature to 20 °C, component c_2 to maintain a given temperature, and, finally, c_3 to ensure the temperature never goes over 20.5 °C. Formally,

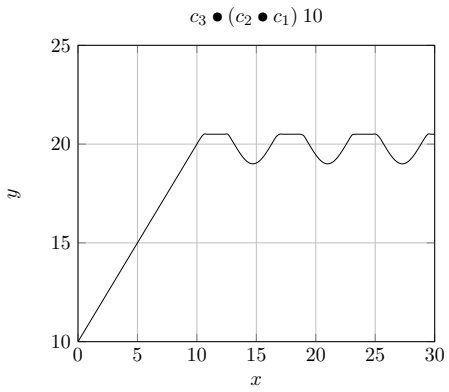
$$
\begin{aligned}
c_1\ x &= \langle\ (x + \_),\ 20 \ominus x\ \rangle \\
c_2\ x &= \langle\ x + (\sin\ \_),\ \infty\ \rangle \\
c_3\ x &= \langle\ \underline{x}\ \lhd\ (x \leq 20.5)\ \rhd\ \underline{20.5}\ , 0\ \rangle
\end{aligned}
$$

where $\ominus$ is truncated subtraction (i.e. $x \ominus y$ is $x - y$ if $x > y$ or 0 otherwise). Composing $c_2 \bullet c_1$ yields a component which reads the current temperature, raises it to 20 °C, and then keeps it stable, as exemplified by the plot below (left). If,

[6] I'll omit the simpler, but similar case dealing with infinite durations.

however, temperatures over 20.5 °C occur, composition $c_3 \bullet (c_2 \bullet c_1)$ puts the system back into the right track as illustrated by the plot in the right.





Clearly, c_3 can be regarded as a supervisor system that, for the sake of efficiency, only acts when temperatures exceed the threshold, using just enough power to keep the temperature below the limit. Actually, note that c_3 is able to play a supervisory role precisely because it is not pre-dynamical.

Recent research [55] unveiled most of the structure of an $\mathcal{H}$-based architectural calculus. In particular, $\mathcal{H}$ is shown to be strong in Top, its Kleisli category to inherit colimits from Top, as expected, and moreover to preserve pullbacks (a little bit harder to prove). A concrete description of the final coalgebra can be done, as well as the systematic definition of old and new combinators for $\mathcal{H}$-based components.

4 Properties

4.1 From invariants to modalities

Requirements, architectural properties, interface specifications, business rules, etc. are common designations for (different kinds of) properties recurring in the practice of Software Engineering. To be unambiguously stated, compared, and even verified against the models of interest, they need to be expressed in suitable logics, preferably equipped with some form of mechanically supported verification framework.

To a great extent, in software design one is interested in properties that are preserved along the system's evolution, the so-called 'business rules', as well as in 'future warranties', stating that *e.g.* some desirable outcome will be eventually produced. Both classes are examples of *modal* assertions, *i.e.* properties that are to be interpreted across a transition system capturing the software dynamics. The relevance of modal reasoning in computing is witnessed by the fact that most university syllabi

in the area include some incursion into modal logic, in particular in its temporal variants.

The novelty is that, as it happens with the notions of transition, behaviour, observational equivalence or refinement, modalities in Coalgebra also acquire a *shape*. That is, their definitions become parametric on whatever type of behaviour seems appropriate for addressing the problem at hand.

Let me start with the notion of an *invariant* – a predicate[7] which is supposed to hold in all states of a system, thus configuring what is classically called a *safety* property. If the system dynamics is described by a coalgebra $\alpha : U \longrightarrow \mathcal{F}(U)$, a predicate ϕ over state space U is an invariant if it holds on the 'current' state and on its 'successor' states, which are of course obtained by execution of α. This entails the need to lift ϕ from U to $\mathcal{F}(U)$.

In a relational setting, *i.e.* regarding predicate ϕ as a coreflexive relation and $\mathcal{F}$ as a relator[8], the informal definition of an invariant can be captured by the statement

$$\forall_{u \in U} \; . \;\; u \; \phi \; u \;\; \Rightarrow \;\; (\alpha \; u) \, \mathcal{F}(\phi) \, (\alpha \; u)$$

which, by eliminating variables, is equivalent to

$$\phi \;\; \subseteq \;\; \alpha^{\circ} \cdot \mathcal{F}(\phi) \cdot \alpha \tag{15}$$

Clearly, just as bisimulations are preserved by the coalgebra transitions, so are invariants. But what is more, the right hand side of expression (15) defines a 'box' modality over the transition system entailed by coalgebra α:

$$\Box\phi \;\; \widehat{=} \;\; \alpha^{\circ} \cdot \mathcal{F}(\phi) \cdot \alpha \tag{16}$$

which rewrites (15) as: ϕ is invariant whenever $\phi \subseteq \Box\phi$. The crucial observation, however, is that the modal operator $\Box$ is parametric on the coalgebra α and, of course, on the functor $\mathcal{F}$. Moreover, invariants induce invariants because, $\Box$ being monotonic,

$$\phi \subseteq \Box\phi \;\; \Rightarrow \;\; \Box\phi \subseteq \Box\Box\phi \; .$$

It is instructive to unfold the definition of the $\Box$ modality for specific cases. Taking $\mathcal{F}(X) = \mathcal{P}(X)$, for example, one gets

$$\Box\phi \;\; = \;\; \{u \in U \mid (\alpha \; u) \; \mathcal{P}(\phi) \; (\alpha \; u)\}$$

[7] In the sequel we will resort, with no change of notation, to two equivalent representations of a predicate over a set X: as a subset of X or as a coreflexive binary relation, *i.e.* a subset of the identity over X. Thus, $x \in \phi$ iff $x \; \phi \; x$.

[8] I've already mentioned relators in footnote 5. The concept of a *relator* [27] extends that of a functor to relations: $\mathcal{F}(R)$ is a relation from $\mathcal{F}(U)$ to $\mathcal{F}(V)$ provided R is a relation from U to V. Relators are monotone and commute with composition, converse and the identity.

which, regarding predicates as sets, takes the more familiar form

$$\Box\phi \ = \{u \in U \mid \alpha\ u \subseteq \phi\}$$

which corresponds to the standard interpretation of the $\Box$ modality in Kripke semantics. As another example consider the functor $\mathcal{F}(X) = \mathbf{1} + X$. Clearly,

$$\Box\phi \ = \ \{u \in U \mid \alpha\ u = \iota_2\ u' \ \Rightarrow \ u' \in \phi\}\ .$$

The whole construction of a modal logic relative to a coalgebra α can be pursued along similar lines. Such a programme is often referred to as the *temporal logic of coalgebras* [41]. Actually, not only a diamond modality is defined, as usual, by duality, $\diamond\phi \mathrel{\widehat{=}} \neg\Box\neg\phi$, but 'temporal extensions' of these modalities can be obtained as fixed points. Consider, for example, the definition of $\boxdot\phi$, the *henceforth* ϕ operator which extends the validity of ϕ over all states computed by successive application of α:

$$\boxdot\phi(x) \quad \widehat{=} \quad \exists_\psi\ .\ \ \psi \text{ is invariant} \ \ \wedge\ \psi \subseteq \phi\ \wedge\ \psi\ x$$

Regarding predicates ϕ and ψ as coreflexives and making explicit the *supremum* implicit in the existential quantification one gets,

$$\begin{array}{rcl}
\boxdot\phi & = & \bigcup \{\psi \mid \psi \subseteq \Box\psi\ \wedge\ \psi \subseteq \phi\} \\
& = & \quad \{\ \cap\text{-universal}\} \\
& & \bigcup \{\psi \mid \psi \subseteq \Box\psi\ \cap\ \phi\} \\
& = & \quad \{\ \text{intersection of coreflexives is relational composition}\ \} \\
& & \bigcup \{\psi \mid \psi \subseteq \phi \cdot \Box\psi\}
\end{array}$$

which leads to a greatest (post)fixed point definition:

$$\boxdot\phi \ = \ \nu\psi\ (\phi \cdot \Box\psi)$$

4.2 Coalgebraic logic

The modalities induced by a coalgebra α and considered so far are relative to the 'global' dynamics of α. Depending on applications, however, one may be interested in other types of modalities. For example, suppose α is a coalgebra for functor $\mathcal{F}(X) = A \times X \times X$. Then, it may be relevant to have modalities to take care of just the right or the left successors.

For another, popular example consider $\mathcal{F}(X) = \mathcal{P}X^A$, the 'shape' of a non-deterministic transition system. In this case one may be interested in one 'box' operator per each action $a \in A$ dealing only with transitions labelled by a. Thus, predicate ϕ over U has to be lifted in a specific way to $\mathcal{P}(U)^A$, for each $a \in A$. The corresponding modality will build on such 'user-defined' lifting.

To proceed, a more general notion of predicate lifting is in order. Fortunately, the definition is straightforward [47]: A predicate lifting is simply a natural transformation $\gamma : \mathbf{2}^- \Longrightarrow \mathbf{2}^{\mathcal{F}(-)}$, where $\mathbf{2}^-$ is the contravariant powerset functor. Then, a modality $\Box$, with respect to a coalgebra $\alpha : U \longrightarrow \mathcal{F}(U)$ and a predicate lifting γ, is defined as

$$\Box \;\widehat{=}\; \mathbf{2}^U \xrightarrow{\gamma_U} \mathbf{2}^{\mathcal{F}(U)} \xrightarrow{\alpha^{-1}} \mathbf{2}^U$$

where f^{-1} denotes the inverse image of function f, *i.e.* $f^{-1}\, Z \;=\; \{u \in U \mid f\, u \in Z\}$[9]. Thus,

$$\Box\phi \;=\; \{u \in U \mid \alpha\; u \in \gamma_U \phi\}\;. \tag{17}$$

For the example above, one specifies a family $\{\gamma^a : \mathbf{2}^- \Longrightarrow \mathbf{2}^{\mathcal{P}(-)^A} \mid a \in A\}$ of predicate liftings

$$\gamma^a_U\; \phi \;\widehat{=}\; \{s \in \mathcal{P}(U)^A \mid s\; a \subseteq \phi\}$$

which induces a corresponding family of $\Box$-like modalities

$$[a]\phi = \{u \in U \mid (\alpha\; u)\; a \subseteq \phi\}\;.$$

As one would expect, those are exactly the indexed modalities of Hennessy–Milner logic [67].

The 'global' modality given by equation (16) in the previous section, can be framed in this more general setting by defining the predicate lifting $\gamma_X\; \phi \;\widehat{=}\; \{s \in \mathcal{F}(X) \mid s\; \mathcal{F}(\phi)\; s\}$.

For the general case one may proceed as follows. As a first step define a signature Σ of modal operators $\divideontimes : X^n \longrightarrow X$, each one with its arity. Then, the syntax of the logic is given by the set of formulas

$$\varphi \;\ni\; p \mid \varphi \wedge \varphi \mid \neg\varphi \mid \divideontimes(\varphi, \cdots, \varphi)$$

for $p \in Prop$, a countable set of propositional variables.

A model M for the logic consists of a coalgebra $\alpha : U \longrightarrow \mathcal{F}(U)$, a valuation $V : Prop \longrightarrow \mathcal{P}(U)$, and, for each n-ary modal symbol $\divideontimes$, an n-ary predicate lifting $\gamma^{\divideontimes} : (\mathbf{2}^-)^n \Longrightarrow \mathbf{2}^{\mathcal{F}(-)}$. Formulas are interpreted over a model inductively: Forgetting

[9] Equivalently, regarding set Z as a function from U to the two element set $\mathbf{2}$, $f^{-1} = \mathbf{2}^f$, with $\mathbf{2}^f\; Z \;=\; Z \cdot f$.

the modal operators for a while, the result is the standard interpretation over the Boolean algebra $\mathcal{P}(U)$. For example, $[\![p]\!] = V(p)$ and $[\![\varphi_1 \wedge \varphi_2]\!] = [\![\varphi_1]\!] \cap [\![\varphi_2]\!]$, as expected. Each n-ary modal operator, on the other hand, is interpreted as

$$[\![\divideontimes(\varphi_1, \cdots, \varphi_n)]\!] \;=\; \alpha^{-1}(\gamma_U^{\divideontimes}([\![\varphi_1]\!], \cdots, [\![\varphi_n]\!]))$$

as in equation (17).

As a final example suppose α is a coalgebra over U for the multiset functor $\mathcal{M}(X) = \mathbb{N}^X$, typically used to capture weighted transition systems. A modal operator $\divideontimes_N$ could be defined to deal with those successor states that are reachable with a cost (measuring *e.g.* resources or time units used) limited to N. Thus,

$$[\![\divideontimes_N\, \varphi]\!] \;=\; \{u \in U \mid \forall_{u' \in U} \,.\, (\alpha\; u)\; u' \leq N \;\Rightarrow\; u' \in [\![\varphi]\!]\}\;.$$

The corresponding predicate lifting is $\gamma_X^{\divideontimes_N}\; \phi \widehat{=} \{s \in \mathbb{N}^X \;\mid \forall_{x \in X} \,.\, s\; x \leq N \Rightarrow x \in \phi\}$. This is similar to what is called in modal logic a *graded* modality, although this qualifier originally refers to a restriction on the cardinality of outgoing transitions from a state, rather than on their weights. Further examples, most useful in software design, are obtained with coalgebras for the distribution monad, for example, to address transitions with a some type of bound on the probability of occurrence.

Of course, the satisfaction relation $\models_M$ for a model M pops out easily. For example,

$$u \models_M \divideontimes_N \varphi \;\;\Leftrightarrow\;\; \forall_{u' \in U} \,.\, (\alpha\; u)\; u' \leq N \;\Rightarrow\; u' \in [\![\varphi]\!]\;.$$

The crucial point is the assignment of a specific predicate lifting to each modal operator in Σ. There is no restriction on how such a lifting is defined but the naturality requirement: This is what ensures that the meaning of the operator will not depend on the state space of the particular coalgebra in a possible model. From the working software engineer perspective, this provides the freedom to define the most suitable logic for the problem at hand.

Such a freedom has an obvious drawback: The definition of the logic along the strategy outlined above is not fully parametric on the functor $\mathcal{F}$. It requires the definition of a set of predicate liftings, one for each modal operator, to give the way in which, for each case, a property over the state space is lifted to an $\mathcal{F}$-structured collection of states. The approach sketched above, however, is the most popular in coalgebraic logic [59]. Actually, it can be formulated in a more abstract setting [48] by first extending the signature Σ to an endofunctor in the category of Boolean algebras, and then interpreting the propositional logic, extended with the operators in Σ, as an algebra for such a functor, *i.e.*

$$\Sigma(\mathbf{2}^U) \xrightarrow{\quad\gamma_U\quad} \mathbf{2}^{\mathcal{F}(U)} \xrightarrow{\quad\alpha^{-1}\quad} \mathbf{2}^U\;.$$

Actually, moving from the powerset Boolean algebra to an arbitrary one is possible because, on extending the propositional calculus, one may always identify propositionally equivalent formulas and equip the corresponding quotient with a Boolean algebra structure.

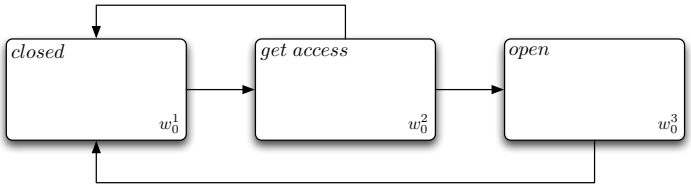
An alternative approach, historically the first to be proposed, builds on L. Moss's original idea [53] of considering functor $\mathcal{F}$ himself as a syntax constructor, therefore leading to a logic which is fully parametric on the functor encoding the system's behaviour. This framework is slightly less general, in the sense that $\mathcal{F}$ is required to preserve weak pullbacks. The main disadvantage, however, from the point of view of Software Engineering applications, is the cumbersome, unintuitive syntax it entails.

In both approaches, however, coalgebraic logic emerges as a powerful, generic theory [25], rather than a way to put together a number of curious examples. The framework is parametric, as discussed, and compositional – a most relevant feature in Computer Science which often requires non trivial combinations of logics. But the hallmark of coalgebraic logic resides in the way most properties one expects to discuss in Logic can be formulated and analysed in this abstract, parametric setting.

A typical example, and most relevant from the applications point of view, concerns the so-called Hennessy–Milner theorem. A modal logic has the Hennessy–Milner property whenever the induced logical equivalence distinguishes between non-bisimilar states and only those. The same applies, in general, to coalgebraic logics. In modal logic, the 'only if' part of the theorem (*i.e.* that logical equivalence entails bisimilarity) requires the underlying Kripke frame to be finitely branching. This is mirrored in the coalgebraic setting through the separability condition which basically says that the logic allows enough predicate liftings to distinguish between all elements in $\mathcal{F}(U)$. The definition of suitable Hilbert calculi as well as the study of expressivity, soundness, completeness and decidability can also be carried out in the abstract setting [59, 47]. In this sense, going coalgebraic seems the right way to do modal logic.

4.3 Illustration: Reasoning about hierarchical designs

Hierarchical transition systems are a popular mathematical structure to represent state-based software applications in which different layers of abstraction are captured by interrelated state machines. The decomposition of high-level states into inner substates, and of their transitions into inner sub-transitions, is a common refinement procedure adopted in a number of specification formalisms. The diagram below depicts a high level behavioural model of a strongbox controller in the form of a transition system with three states.



The strongbox can be open, closed, or going through an authentication process. The model can be formalised in some sort of modal logic, so that state transitions can be expressed, possibly combined with hybrid features to refer to specific, individual states. The qualifier *hybrid* [15] refers to an extension of modal languages with symbols, called *nominals*, which explicitly refer to individual states in the underlying Kripke frame[10]. A satisfaction operator $@_i\varphi$ stands for φ holding in the state named by nominal i.

For example, in propositional hybrid logic [23] and assuming

$$\text{Nom} = \{closed, get\ access, open\}$$

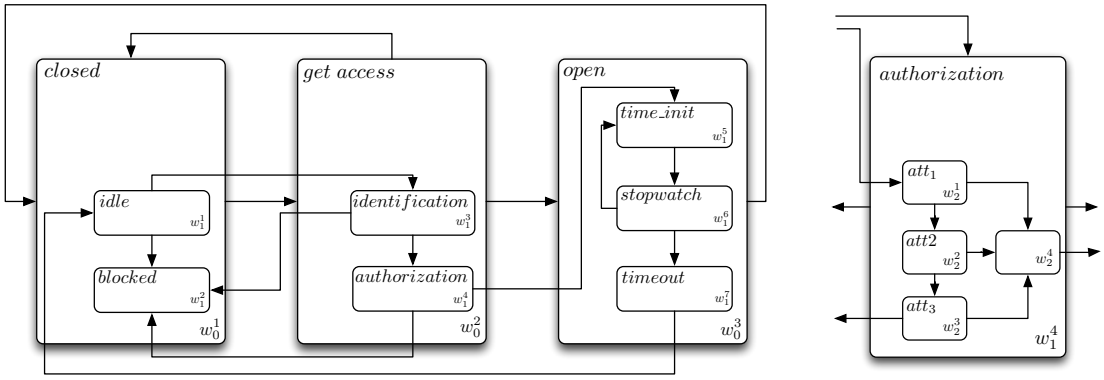
as a set of nominals, a number of properties of the the diagram above can be expressed, *e.g.*

- the state *get access* is accessible from the state *closed*: $@_{closed}\Diamond get\ access$,
- or the state *open* is not directly accessible from *closed*: $\Diamond open \rightarrow \neg closed$.

This high level vision of the strongbox controller can be refined by decomposing not only its internal states, but also its transitions. Thus, each 'high-level' state gives rise to a new, local transition system, and each 'upper-level' transition is decomposed into a number of 'intrusive' transitions from sub-states of the 'lower-level' transition system corresponding to the refinement of the original source state, to sub-states of the corresponding refinements of original target states. For instance, the (upper) *close* state can be refined into a (inner) transition system with two (sub) states: one, *idle*, representing the system waiting for the order to proceed for the *get access* state, and another one, *blocked*, capturing a system which is unable to proceed with the opening process (e.g. when authorised access for a given user was definitively denied). In this scenario, the upper level transition from *closed* to *get access* can be realised by, at least, one intrusive transition between the *closed* sub-state *idle* and the *get access* sub-state $identification$, in which the user identification is to be checked before proceeding. This refinement step is illustrated in the diagram below

[10]Notice the same adjective was used in the previous sections with a totally different meaning: to refer to software components with both discrete and continuous behaviour. The designation *cyber-physical* is also used in the later case with a similar meaning.

(left). Still the specifier may go even further. For example, she may like to refine the *get access* sub-state *authorisation* into the more fine-grained transition structure depicted on the right hand side of the diagram. This third-level view includes a sub-state corresponding to each one of the possible three attempts of password validation, as well as an auxiliary state to represent authentication success.



Such a hierarchical way to design a system is quite natural and somehow inherent to well-known design formalisms such as D. Harel's statecharts [29] and the subsequent UML hierarchical state machines, among others. This sort of systems have been studied in my own research group in the context of reconfigurable software architectures [49]. In particular, a hierarchical hybrid logic was proposed to express (and reason about) requirements that typically involve transitions between designated states in different local transition systems.

The whole programme can actually be carried out in a coalgebraic setting. The first observation is that a measure of maturity of coalgebraic logic is its ability to incorporate extensions which are already classical in the modal logic literature. We have briefly mentioned how new 'temporal' operators can be defined through fixed points, as in the modal μ-calculus [46]. Hybrid logic is also easily accommodated in a way which is quite similar to what is done in modal logic with classical Kripke semantics. In particular, the set of formulas is extended with

$$\varphi \ni \cdots \mid i \mid @_i \varphi$$

for $i \in Nom$, a set of nominals. The original valuation is extended to $V : Prop \cup Nom \longrightarrow \mathcal{P}(U)$ with the restriction that $V\ i$ is a singleton for each nominal i, *i.e.* a nominal identifies a unique state in the state space. The interpretation of the hybrid operators is the classical one: $[\![@_i\varphi]\!] = \{u \in U \mid V\ i \in [\![\varphi]\!]\}$ and $[\![i]\!] = V\ i$. The only aspect one needs to take into account is the interplay between the satisfaction

operators and the modalities induced (or built over) the coalgebra. For example, one has to specify that a formula like $@_i\ \varphi$ must be valid either in the whole model or nowhere. In an Hilbert calculus this can be achieved through an extra axiom, for each modal operator $\divideontimes$:

$$@_i\varphi \ \Rightarrow\ (\divideontimes(\varphi_1, \cdots, \varphi_k)\ \Leftrightarrow\ \divideontimes(\varphi_1 \wedge @_i\varphi, \cdots, \varphi_k \wedge @_i\varphi)$$

capturing the intended validity of $@_i\varphi$ irrespective to the interpretation of each φ_j.

In the definition of a model for this logic the family of accessibility relations considered in [49] is replaced by a family of coalgebras for the same endofunctor, each of which captures the dynamics of the appropriate layer.

Signatures are n-families of disjoint, possible empty, sets of symbols

$$\Delta^n = (\mathrm{Prop}_k, \mathrm{Nom}_k)_{k\in\{0,\cdots,n\}}\ .$$

For example, to specify the strongbox above, one considers a signature Δ^2 for the three layers presented. 0-level symbols consist of the set of nominals

$$\mathrm{Nom}_0 = \{closed_0, get\_access_0, open_0\}$$

and a set of propositions Prop_0. The 1-level signature introduces a set of nominals

$$\begin{aligned}\mathrm{Nom}_1 = \{&idle_1, blocked_1, identification_1, authorization_1, time\_init_1,\\ &stopwatch_1, time\_out_1\}\end{aligned}$$

and, again, a set of propositions Prop_1. Level 3, finally, introduces att_{12}, att_{22} and att_{32} in Nom_2. The set of formulas $Fm(\Delta^n)$ is the n-family recursively defined, for each k, by

$$\varphi_0 \ni i_0 \mid p_0 \mid \neg\varphi_0 \mid \varphi_0 \wedge \varphi_0 \mid @_{i_0}\varphi_0 \mid \Box_0\varphi_0$$
$$\varphi_0^b \ni i_0 \mid p_0 \mid @_{i_0}\varphi_0 \mid \Box_0\varphi_0$$

where superscript b qualifies the basic formulas, and

$$\varphi_k \ni \varphi_{k-1}^b \mid i_k \mid p_k \mid \neg\varphi_k \mid \varphi_k \wedge \varphi_k \mid @_{i_k}\varphi_k \mid \ \diamond_k\ \varphi_k$$

where for any $k \in \{1, \ldots, n\}$, the basic formulas are defined by

$$\varphi_{k-1}^b \ni i_{k-1} \mid p_{k-1} \mid \varphi_{k-2}^b \mid @_{i_{k-1}}\varphi_{k-1} \mid \Box_{k-1}\varphi_{k-1}$$

for $k \in \{2, \cdots, n\}$, $p_k \in \mathrm{Prop}_k$ and $i_k \in \mathrm{Nom}_k$.

This language is able to express quite different properties. For instance, inner-outer relations between named states, e.g. $@_{idle_1}closed_0$ or $@_{att_{12}}open_0$, as well

as a variety of transitions. Those include, for example, the layered transition $@_{get\_access_0} \diamond_0 open_0$, the 0-internal transition $@_{identification_1} \diamond_1 authorisation_1$ or intrusive transitions like $@_{idle_1} \diamond_1 authorisation_1$ and $get\_access_0 \to \diamond_1 open_0$.

The definition of a model is parameterised by a family of coalgebras defined for the same functor, *i.e.* exhibiting the same type of behavioural effect. A n-layered model $M \in \mathrm{Mod}^n(\Delta^n)$ is a tuple

$$M = \langle W^n, D^n, \alpha^n, V^n \rangle$$

where $W^n = (W_k)_{k\in\{0,\cdots,n\}}$ is a family of disjoint sets of states, and $D^n \subseteq W_0 \times \cdots \times W_n$ is a definition predicate that singles out the chains of states across the n levels which are considered meaningful 'global' states. Denoting by D_k the k-restriction $D^n|_k$ to the first $k+1$ columns, for each $k \in \{0, \cdots, n\}$, it is the case that

$$W_k = \{v_k | D_k\langle w_0, \cdots, w_{k-1}, v_k\rangle, \text{ for some } w_0, \cdots, w_{k-1} \text{ st } D_{k-1}\langle w_0, \cdots, w_{k-1}\rangle\} \ .$$

Then, comes the 'dynamics': $\alpha^n = (\alpha_k : D_k \longrightarrow \mathcal{F}(D_k))_{k\in\{0,\cdots,n\}}$ is a family of $\mathcal{F}$-coalgebras specifying the system's evolution at each level in the hierarchy. Finally, $V^n = (V_k^{\mathrm{Prop}}, V_k^{\mathrm{Nom}})_{k\in\{0,\cdots,n\}}$ is a family of pairs of valuations defined as one could expect:

- $V_k^{\mathrm{Prop}} : \mathrm{Prop}_k \to \mathcal{P}(D_k)$, and
- $V_k^{\mathrm{Nom}} : \mathrm{Nom}_k \to W_k$.

The satisfaction relation takes a similar shape as a family of relations

$$\models^n = \ (\models_k)_{k\in\{0,\cdots,n\}}$$

defined, for each $w_r \in W^r$, $r \in \{0, \cdots, k\}$, $k \leq n$, such that $D_k\langle w_0, \cdots w_k\rangle$. The case of interest in the context of this paper is the one for modalities, *i.e.* $M_k, w_0, \cdots, w_k \models_k \Box_k \varphi_k$ iff

$$\forall_{v_0\in W_0,\cdots,v_k\in W_k}.\ \langle v_0, \cdots, v_k\rangle \in \alpha_k\langle w_0, \cdots, w_k\rangle \text{ implies } M, v_0, \cdots, v_k \models_k \varphi_k \ .$$

The hybrid part is given by

- $M_k, w_0, \cdots, w_k \models_k i_k$ iff $w_k = V_k^{\mathrm{Nom}}(i_k)$ and $D_k\langle w_0, \cdots, w_{k-1}, V_k^{\mathrm{Nom}}(i_k)\rangle$,
- $M_k, w_0, \cdots, w_k \models_k @_{i_k}\varphi_k$ iff $M_k, w_0, \cdots w_{k-1}, V_k^{\mathrm{Nom}}(i_k) \models_k \varphi_k$ and $D_k\langle w_0, \cdots w_{k-1}, V_k^{\mathrm{Nom}}(i_k)\rangle$.

The Boolean part, finally, is defined as usual, just taking care of the definability interdependence captured by D^n. Thus,

- $M_k, w_0, \cdots, w_k \models_k \varphi^b_{k-1}$ iff $M_{k-1}, w_0, \cdots, w_{k-1} \models_{k-1} \varphi^b_{k-1}$,
- $M_k, w_0, \cdots, w_k \models_k p_k$ iff $\langle w_0, \cdots, w_k \rangle \in V^{\text{Prop}}_k(p_k)$,
- $M_k, w_0, \cdots, w_k \models_k \varphi_k \wedge \varphi'_k$ iff $M_k, w_0, \cdots, w_k \models_k \varphi_k$ and $M_k, w_0, \cdots, w_k \models_k \varphi'_k$,
- $M_k, w_0, \cdots, w_k \models_k \neg\varphi_k$ iff it is false that $M_k, w_0, \cdots, w_k \models_k \varphi_k$.

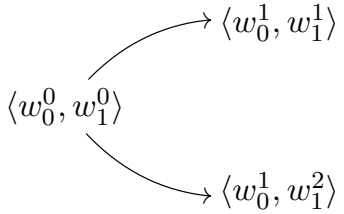
The resulting logic is quite expressive. Notions of n-layered bisimilarity and refinement can be introduced [49] along the lines already discussed in this paper, and a Hennessy–Milner theorem proved.

A specific, particularly well-behaved class of layered models, is called *hierarchical*: it requires that the restriction of a coalgebra α_k to the state space of α_{k-1} coincides with the latter. This ensures that the elements in the family of coalgebras are compatible.

The example sketched here is clearly an hierarchical model. Examples of non-hierarchical layered models can be achieved by removing some 0-transitions depicted in the diagram above (e.g. the one linking the named states $closed_0$ and get_access_0). This *hierarchical* condition can be expressed as a naturality condition as follows. Define $\pi_k : D_k \longrightarrow D_{k-1}$ by $\pi_k \langle w_0, \cdots, w_{k-1}, w_k \rangle \mathrel{\widehat{=}} \langle w_0, \cdots, w_{k-1} \rangle$. Then, the model is hierarchical if, for all k, the following diagram commutes[11].

$$\begin{array}{ccc} D_k & \xrightarrow{\alpha_k} & \mathcal{F}(D_k) \\ {\scriptstyle \pi_k}\downarrow & & \downarrow{\scriptstyle \mathcal{F}(\pi_k)} \\ D_{k-1} & \xrightarrow{\alpha_{k-1}} & \mathcal{F}(D_{k-1}) \end{array}$$

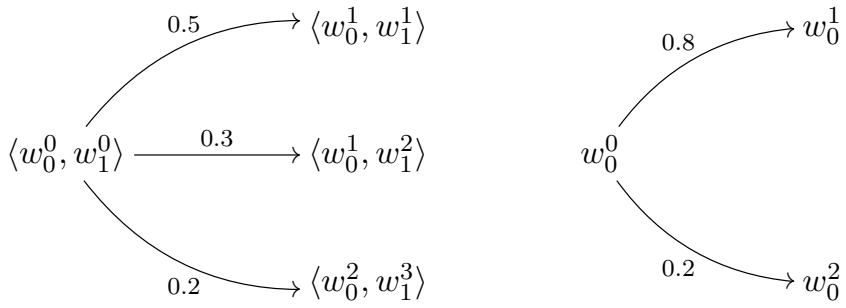
For $\mathcal{F} = \mathcal{P}$ this means, for example, that the transitions depicted in the diagram



exist at level 1 iff a transition $w^0_0 \longrightarrow w^1_0$ exists at level 0.

[11] This basically means that the family α^n of coalgebras in a model of a hierarchical system, can be regarded as a coalgebra in the category of pre-sheaves $[\overline{n}, Set]$, where $\overline{n}$ is the total order corresponding to the initial n-segment of natural numbers. Such a coalgebra is, of course, a natural transformation.

As another example consider $\mathcal{F} = \mathcal{D}$. In an hierarchical (probabilistic) system the 1-level transitions in the left of the diagram below exist if the 0-level transitions depicted on the right exist as well.



5 Concluding

This paper revisited a few themes in elementary, *i.e. Set*-based, Coalgebra in connection with what may be regarded as the kernel activities of a software engineer: *modelling* complex systems, *architecting* their composition and *reasoning* about their behaviour. Models, architectures and properties were therefore the buzzwords chosen to guide this exercise.

As a design discipline, Software Engineering is currently challenged by continuous technological evolution towards very large, heterogeneous, highly dynamic computing systems, which require innovative approaches to master their complexity. Systems whose behaviour cannot be simply characterised in terms of a relation between input and output data, but expresses a continued interaction with their external (computational or physical) and internal (sub-systems) contexts. In this sense, they can be classified as *reactive*, to use a term coined by A. Pnueli and D. Harel [30] in the 1980s. Furthermore, concurrent composition is the norm, rather than the exception.

Developing such systems correctly is very difficult, because it involves not only mastering the complexity of building and deploying large applications on time and within budget, but also managing an open-ended structure of autonomous components, typically distributed, often organised in loosely coupled configurations, and highly heterogeneous. Additional difficulties arise with the need to take into account a plethora of issues such as real-time responsiveness, dynamic reconfiguration, QoS-awareness, self-adaptability, security, dependability, under-specification of third-party components, among many others.

Unfortunately, software technology is still pre-scientific in its lack of sound mathematical foundations to provide an effective basis to predict and certify computa-

tionally generated behaviour. In a sense, compared to other Engineering disciplines, we are just living our 17th century, seeking for the right foundations, methods and calculi to move from *ad hoc* to systematic and accountable engineering practices.

My purpose was to explore one of those mathematical frameworks which has the potential to address a large class of computational systems. Indeed, we would suggest Coalgebra as a, probably *the*, mathematics for dynamical, state-based systems.

The essence of the coalgebraic method boils down to a very basic observation: that from a suitable characterisation of the *type* of a system's dynamics, canonical notions of behaviour, observational reasoning (equational and inequational) and modality can be derived in a uniform (*i.e.* parametric) way.

This setting may sound familiar to the working software engineer: the object of her practice, if not of her study, is precisely the ubiquity of the computing phenomena along and across universes of typed arrows. Arrows may stand for functions, algorithms, services or components, programs fulfilling a specification contract, relationships in a UML diagram, processes through mobile ambients, evolutions in a sensor network, links in a software architectural description, circuits coordinating loosely coupled agents, or whatever structures our domain. The type of a coalgebra, an endofunctor, is itself an arrow, and so is, moreover, the coalgebra itself.

In a brief historical overview of the trends and results predating the emergence of Coalgebra in Computer Science, B. Jacobs [41], referring to the work of Arbib, Manes, Goguen, Adamek and others in the late 1970s, on categorical approaches to systems theory, comments: *Their aim was to place sequential machines and control systems in a unified framework* which *(...) led to general notions of state, behaviour, reachability, observability, and realisation of behaviour.* Jacobs remarks, however, that the reason why Coalgebra did not emerge directly from this work was *probably because the setting of modules and vector spaces from which this work arose provided too little categorical infrastructure (especially: no cartesian closure).* The quick expansion of Coalgebra, its techniques and applications, and the capacity shown to capture in a uniform, parametric way a myriad of state-based systems, as well as its mathematical elegance, offers evidence we may be on the right track.

The development of Coalgebra and its application to Computer Science stemmed from different sources, from P. Aczel's non well-founded set theory, accommodating infinitely descending $\in$-chains, to the study of infinite data types and the theory of behavioural specification [40] and satisfaction [14]. Still, it remains an area of active research, with a growing impact not only on the foundations of computing semantics, but also on very concrete programming techniques. Actually, there is a growing interest on the potential of coalgebraic techniques in algorithm understanding and derivation, often based on rediscovering and generalising specific algorithms, for example from automata theory [21, 17, 22, 26]. Unsurprisingly, going generic in the

theory often leads to efficient computational solutions. Striking developments on coinductive proof methods, notably the recent work on up-to techniques [18, 19], go in a similar direction.

Without trying to be exhaustive, we would still like to mention a few other current research directions which will certainly have an impact in the coming decade. The first concerns the combination of algebraic and coalgebraic techniques and the discovery of compatible patterns described by distributive laws [45], which, as shown in D. Turi and J. J. M. M. Rutten landmark work, in the late 1990's, correspond to specification formats in operational semantics. The impact of such laws in several constructions, for example in the formulation of trace semantics, as mentioned above, but also in combining monadic and comonadic effects [12] and logic, suggests we are dealing with some sort of very fundamental structures.

Another direction addresses the challenge of quantitative (weighted, probabilistic, continuous) reasoning, once again driven by the broadening spectrum of Software Engineering problems. This is not only pushing the development of Coalgebra within categories different from *Set* [16, 44, 55], but also leading a lot of results on behavioural *metrics* as an alternative to equivalences [24, 3, 5]. Actually, in the context of *e.g.* probabilistic or hybrid systems, working with equivalences entailing the need for exact matching of real numbers is unrealistic. Metrics, on the other hand, can measure how close two systems are and conclude whether they should be taken as equivalent.

But the impact of Coalgebra can also be recognised at a more 'syntactical' level. The work of A. Silva, a former student of this University, on the derivation of specification languages from the functor typing the coalgebra dynamics [64, 63] should be mentioned here. In a more general setting, the points of contact between Coalgebra and current research on graphical languages in which diagrams, syntax and interpretations, are generated as arrows in special families of monoidal categories [71, 20], seem most promising, with applications ranging from the 're-interpretation' of classical control theory to the design of diagrammatic languages to express, *e.g.*, software architectures.

I'm not sure whether this paper was able to raise the interest of the working software engineer in Coalgebra, or, on the other hand, that of the logician who may find in Computer Science a huge domain for the fruitful application of her methods and tools. In 1967, Anthony Oettinger [56], speaking as President of the ACM, recognised that the expression Software Engineering

> *seems strange to classical engineers and to classical mathematicians alike, because, you know, why would a mathematician think of engineering with symbols and, by the same token, why would somebody who thinks of engineering in terms of things we do with pieces of metal or transistors, think of an operation that takes place on paper with pencils and erasers as engineering.*

Almost 50 years later, there is still a need to push back this discipline to where it actually belongs. Fortunately, to continue with A. Oettinger's speech, *there is no question but that the study of symbol systems, of effective algorithms, of efficient algorithms, of the structure of algorithms, is a mathematical discipline.* And, moreover, *there is, in this realm, enough elegance to attract anybody who wants a challenge.*

Doing Software Engineering in lighter, more informal ways, brings to my mind a quotation attributed to Vlad Patryshev in a slightly different context: *It's like talking about electricity without using calculus. Good enough to replace a fuse, not enough to design an amplifier.*

Acknowledgements. Sections 2.3, 3.2 and 4.3 illustrate, through three concrete applications, the potential Coalgebra may have for the working software engineer with respect to each of the topics chosen for this paper: *models*, *architectures* and *properties*. These applications come from current research along which I had the privilege of collaborating with a number of colleagues. In particular, the coalgebraic treatment of hybrid systems was developed by Renato Neves, who introduced the $\mathcal{H}$ monad and a number of exciting results still emerging at the time of writing. The remaining 'illustrations' are also in debt to ongoing collaboration with José Nuno Oliveira, on formal approaches to software architecture, Sun Meng, on coalgebraic refinement, and both Alexandre Madeira and Manuel A. Martins, on hybrid logics for reconfigurable systems.

References

[1] P. Aczel. *Non-Well-Founded Sets.* CSLI Lecture Notes (14), Stanford, 1988.

[2] P. Aczel and N. Mendler. A final coalgebra theorem. In D. Pitt, D. Rydeheard, P. Dybjer, A. Pitts, and A. Poigne, editors, *Proc. Category Theory and Computer Science*, pages 357–365. Springer Lect. Notes Comp. Sci. (389), 1988.

[3] G. Bacci, G. Bacci, K. G. Larsen, and R. Mardare. Computing behavioral distances, compositionally. In Krishnendu Chatterjee and Jirí Sgall, editors, *Mathematical Foundations of Computer Science 2013 - 38th International Symposium, MFCS 2013,*

Klosterneuburg, Austria, August 26-30, 2013., pages 74–85. Springer Lect. Notes Comp. Sci. (8087), 2013.

[4] J. Baez and J. Dolan. Categorification. In Ezra Getzler and Mikhail Kapranov, editors, *Higher Category Theory*, Contemp. Math. 230, pages 1–36. American Mathematical Society, 1998.

[5] P. Baldan, F. Bonchi, H. Kerstan, and B. König. Behavioral metrics via functor lifting. In Venkatesh Raman and S. P. Suresh, editors, *34th International Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS 2014, December 15-17, 2014, New Delhi, India*, volume 29 of *LIPIcs*, pages 403–415. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2014.

[6] L. S. Barbosa. *Components as Coalgebras.* PhD thesis, DI, Universidade do Minho, 2001.

[7] L. S. Barbosa. Towards a calculus of state-based software components. *"Jour. Universal Comp. Sci."*, 9(8):891–909, 2003.

[8] L. S. Barbosa and J. N. Oliveira. Transposing partial components: an exercise on coalgebraic refinement. *Theor. Comp. Sci.*, 365(1-2):2–22, 2006.

[9] L. S. Barbosa, J. N. Oliveira, and A. M. Silva. Calculating invariants as coreflexive bisimulations. In J. Meseguer and G. Rosu, editors, *Algebraic Methodology and Software Technology, 12th International Conference, AMAST 2008, Urbana, IL, USA, July 28-31, 2008, Proceedings*, pages 83–99. Springer Lect. Notes Comp. Sci. (5140), 2008.

[10] L. S. Barbosa, M. Sun, B. K. Aichernig, and N. Rodrigues. On the semantics of componentware: a coalgebraic perspective. In Jifeng He and Zhiming Liu, editors, *Mathematical Frameworks for Component Software: Models for Analysis and Synthesis*, Series on Component-Based Software Development, pages 69–117. World Scientific, 2006.

[11] J. Beaten and W. Weijland. *Process Algebra.* Cambridge University Press, 1990.

[12] M. Behrisch, S. Kerkhoff, and J. Power. Category theoretic understandings of universal algebra and its dual: Monads and lawvere theories, comonads and what? *Electr. Notes Theor. Comput. Sci.*, 286:5–16, 2012.

[13] J. Benabou. Introduction to bicategories. *Springer Lect. Notes Maths. (47)*, pages 1–77, 1967.

[14] M. Bidoit and R. Hennicker. Behavioural theories and the proof of behavioural properties. *Theor. Comput. Sci.*, 165(1):3–55, 1996.

[15] P. Blackburn. Representation, reasoning, and relational structures: a hybrid logic manifesto. *Logic Journal of IGPL*, 8(3):339–365, 2000.

[16] F. Bonchi, M. M. Bonsangue, M. Boreale, J. J. M. M. Rutten, and A. Silva. A coalgebraic perspective on linear weighted automata. *Inf. Comput.*, 211:77–105, 2012.

[17] F. Bonchi, M. M. Bonsangue, H. H. Hansen, P. Panangaden, J. J. M. M. Rutten, and A. Silva. Algebra-coalgebra duality in Brzozowski's minimization algorithm. *ACM Trans. Comput. Log.*, 15(1):3, 2014.

[18] F. Bonchi, D. Petrisan, D. Pous, and J. Rot. Coinduction up-to in a fibrational setting.

In Thomas A. Henzinger and Dale Miller, editors, *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*, pages 20:1–20:9. ACM, 2014.

[19] F. Bonchi, D. Petrisan, D. Pous, and J. Rot. Lax bialgebras and up-to techniques for weak bisimulations. In Luca Aceto and David de Frutos-Escrig, editors, *26th International Conference on Concurrency Theory, Madrid, September 1.4, 2015*, volume 42 of *LIPIcs*, pages 240–253. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[20] F. Bonchi, P. Sobocinski, and F. Zanasi. Full abstraction for signal flow graphs. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 515–526. ACM, 2015.

[21] Filippo Bonchi and Damien Pous. Checking NFA equivalence with bisimulations up to congruence. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 457–468. ACM, 2013.

[22] Filippo Bonchi and Damien Pous. Hacking nondeterminism with induction and coinduction. *Commun. ACM*, 58(2):87–95, 2015.

[23] T. Brauner. *Hybrid Logic and its Proof-Theory.* Applied Logic Series. Springer, 2010.

[24] F. van Breugel and J. Worrell. Approximating and computing behavioural distances in probabilistic transition systems. *Theor. Comput. Sci.*, 360(1-3):373–385, 2006.

[25] C. Cîrstea, A. Kurz, D. Pattinson, L. Schröder, and Y. Venema. Modal logics are coalgebraic. *Comput. J.*, 54(1):31–41, 2011.

[26] N. Foster, D. Kozen, M. Milano, A. Silva, and L. Thompson. A coalgebraic decision procedure for netkat. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 343–355. ACM, 2015.

[27] P. J. Freyd and A. Ščedrov. *Categories, Allegories*, volume 39 of *Mathematical Library.* North-Holland, 1990.

[28] E. Haghverdi, P. Tabuada, and G. J. Pappas. Bisimulation relations for dynamical, control, and hybrid systems. *Theoretical Computer Science*, 342(2-3):229–261, 2005.

[29] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[30] D. Harel and A. Pnueli. On the development of reactive systems. In *Logics and Models of Concurrent Systems*, volume 13 of *NATO Adv. Sci. Inst. Ser. F Comput. Systems Sci.*, pages 477–498. Springer-Verlag, 1985.

[31] I. Hasuo. Generic forward and backward simulations. In Christel Baier and Holger Hermanns, editors, *CONCUR 2006 - Concurrency Theory, 17th International Conference, CONCUR 2006, Bonn, Germany, August 27-30, 2006, Proceedings*, volume 4137 of *Lecture Notes in Computer Science*, pages 406–420. Springer, 2006.

[32] I. Hasuo. The microcosm principle and compositionality of GSOS-based component calculi. In Andrea Corradini, Bartek Klin, and Corina Cîrstea, editors, *Algebra and Coalgebra in Computer Science - 4th International Conference, CALCO 2011, Winchester, UK, August 30 - September 2, 2011.*, pages 222–236. Springer Lect. Notes Comp. Sci. (6859), 2011.

[33] I. Hasuo, C. Heunen, B. Jacobs, and A. Sokolova. Coalgebraic components in a many-sorted microcosm. In Alexander Kurz, Marina Lenisa, and Andrzej Tarlecki, editors, *Algebra and Coalgebra in Computer Science, Third International Conference, CALCO 2009, Udine, Italy, September 7-10, 2009. Proceedings*, pages 64–80. Springer Lect. Notes Comp. Sci. (5728), 2009.

[34] I. Hasuo and B. Jacobs. Traces for coalgebraic components. *Mathematical Structures in Computer Science*, 21(2):267–320, 2011.

[35] I. Hasuo, B. Jacobs, and A. Sokolova. Generic trace semantics via coinduction. *Logical Methods in Computer Science*, 3(4), 2007.

[36] T. A. Henzinger. The theory of hybrid automata. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30, 1996*, pages 278–292. IEEE Computer Society, 1996.

[37] C. A. R. Hoare, S. van Staden, B. Möller, G. Struth, J. Villard, H. Zhu, and P. W. O'Hearn. Developments in concurrent kleene algebra. In Peter Höfner, Peter Jipsen, Wolfram Kahl, and Martin Eric Müller, editors, *Relational and Algebraic Methods in Computer Science - 14th International Conference, RAMiCS 2014, Marienstatt, Germany, April 28-May 1, 2014.*, pages 1–18. Springer Lect. Notes Comp. Sci. (8428), 2014.

[38] P. F. Hoogendijk. *A generic theory of datatypes*. PhD thesis, Department of Computing Science, Eindhoven University of Technology, 1996.

[39] J. Hughes. Generalising monads to arrows. *Sci. Comput. Program.*, 37(1-3):67–111, 2000.

[40] U. L. Hupbach and H. Reichel. On behavioural equivalence of data types. *Elektronische Informationsverarbeitung und Kybernetik*, 19(6):297–305, 1983.

[41] B. Jacobs. *Introduction to Coalgebra. Towards Mathematics of States and Observations*. Cambridge University Press (to appear), 2012. Draft copy: Version 2.0, 2012. Institute for Computing and Information Sciences, Radboud University Nijmegen.

[42] B. Jacobs, A. Silva, and A. Sokolova. Trace semantics via determinization. *J. Comput. Syst. Sci.*, 81(5):859–879, 2015.

[43] A. Joyal, R. Street, and D. Verity. Traced monoidal categories. *Math. Proc. Camb. Phil. Soc.*, 119:447–468, 1996.

[44] H. Kerstan and B. König. Coalgebraic trace semantics for continuous probabilistic transition systems. *Logical Methods in Computer Science*, 9(4), 2013.

[45] B. Klin. Bialgebras for structural operational semantics: An introduction. *Theor. Comput. Sci.*, 412(38):5043–5069, 2011.

[46] D. Kozen. Results on the propositional mu-calculus. *Theor. Comput. Sci.*, 27:333–354,

1983.

[47] C. Kupke and D. Pattinson. Coalgebraic semantics of modal logics: An overview. *Theor. Comput. Sci.*, 412(38):5070–5094, 2011.

[48] A. Kurz and R. L. Leal. Modalities in the stone age: A comparison of coalgebraic logics. *Theor. Comput. Sci.*, 430:88–116, 2012.

[49] A. Madeira, M. A. Martins, and L. S. Barbosa. A logic for n-dimensional hierarchical refinement. In John Derrick, Eerke A. Boiten, and Steve Reeves, editors, *Proceedings 17th International Workshop on Refinement, Refine@FM 2015, Oslo, Norway, 22nd June 2015*, volume 209 of *EPTCS*, pages 40–56, 2016.

[50] R. Milner. Elements of interaction (Turing Award Lecture). *Communications of the ACM*, 36(1):78–89, 1993.

[51] R. Milner. *Communicating and Mobile Processes: the π-Calculus*. Cambridge University Press, 1999.

[52] V. C. Miraldo and J. N. Oliveira. 'keep definition, change category' – a practical approach to state-based system calculi. *Journal of Logical and Algebraic Methods in Programming*, 85(4):449–474, 2016.

[53] L. Moss. Coalgebraic logic. *Ann. Pure & Appl. Logic*, 96(1-3):277–317, 1999.

[54] R. Neves and L. S Barbosa. Hybrid automata as coalgebras. In Augusto Sampaio and Farn Wang, editors, *Theoretical Aspects of Computing - ICTAC 2016 - 13th International Colloquium, Taiwan, Proceedings*, pages 385–402. Springer Lect. Notes Comp. Sci. (9965), 2016.

[55] R. Neves, L. S. Barbosa, D. Hofmann, and M. A. Martins. Continuity as a computational effect. *J. Log. Algebr. Meth. Program.*, 85(5):1057–1085, 2016.

[56] A. G. Oettinger. The hardware-software complementarity. *Commun. ACM*, 10(10):604–606, 1967.

[57] J. N. Oliveira. Towards a linear algebra of programming. *Formal Aspects of Computing*, 24(4-6):433–458, 2012.

[58] D. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, *Proc. Conf. on Theoretical Computer Science*, pages 167–183. Springer Lect. Notes Comp. Sci. (104), 1981.

[59] D. Pattinson. Coalgebraic modal logic: soundness, completeness and decidability of local consequence. *Theor. Comput. Sci.*, 309(1-3):177–193, 2003.

[60] E. P. Robinson. Variations on algebra: monadicity and generalisations of equational theories. *Formal Asp. Comput.*, 13(3-5):308–326, 2002.

[61] J. J. M. M. Rutten. Universal coalgebra: A theory of systems. *Theor. Comput. Sci.*, 249(1):3–80, 2000. (Revised version of CWI Techn. Rep. CS-R9652, 1996).

[62] D. Sangiorgi. On the origins of bisimulation and coinduction. *ACM Trans. Program. Lang. Syst.*, 31(4), 2009.

[63] A. Silva, F. Bonchi, M. Bonsangue, and J. J. M. M. Rutten. Quantitative Kleene coalgebras. *Inf. Comput.*, 209(5):822–849, 2011.

[64] A. Silva, M. M. Bonsangue, and J. J. M. M. Rutten. Non-deterministic Kleene coalge-

bras. *Logical Methods in Computer Science*, 6(3), 2010.

[65] M. Smyth and G. Plotkin. The category theoretic solution of recursive domain equations. *SIAM Journ. Comput.*, 4(11):761–783, 1982.

[66] A. Sokolova. Probabilistic systems coalgebraically: A survey. *Theor. Comput. Sci.*, 412(38):5095–5110, 2011.

[67] C. Stirling. Modal and temporal logics for processes. *Springer Lect. Notes Comp. Sci. (715)*, pages 149–237, 1995.

[68] M. Sun and L. S. Barbosa. Components as coalgebras: The refinement dimension. *Theor. Comp. Sci.*, 351:276–294, 2006.

[69] N. Urabe and I. Hasuo. Generic forward and backward simulations III: quantitative simulations by matrices. In Paolo Baldan and Daniele Gorla, editors, *CONCUR 2014 - Concurrency Theory - 25th International Conference, CONCUR 2014, Rome, Italy, September 2-5, 2014.*, pages 451–466. Springer Lect. Notes Comp. Sci. (8704), 2014.

[70] N. Urabe and I. Hasuo. Coalgebraic infinite traces and kleisli simulations. In Lawrence S. Moss and Pawel Sobocinski, editors, *6th Conference on Algebra and Coalgebra in Computer Science, CALCO 2015, June 24-26, 2015, Nijmegen, The Netherlands*, volume 35 of *LIPIcs*, pages 320–335. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[71] Fabio Zanasi. *Interacting Hopf Algebras- the Theory of Linear Systems*. PhD thesis, École normale supérieure de Lyon, France, 2015.

Received 02 March 2020