A calculus for generic, QoS-aware component composition

L. S. Barbosa and Sun Meng

Abstract. Software QoS properties, such as response time, availability, bandwidth requirement, memory usage, among many others, play a major role in the processes of selecting and composing software components. This paper extends a component calculus to deal, in an effective way, with them. The calculus models components as generalised Mealy machines, *i.e.*, state-based entities interacting along their life time through well defined interfaces of observers and actions. QoS is introduced through an algebraic structure specifying the relevant QoS domain and how its values are composed under different disciplines. A major effect of introducing QoS-awareness is that a number of equivalences holding in the plain calculus become refinement laws. The paper also introduces a prototyper for the calculus developed as a 'proof-of-concept' implementation.

Mathematics Subject Classification (2000). Primary 68N30; Secondary 68Q85.

Keywords. Coalgebra, component, QoS, refinement.

1. Introduction

Component-based software development [24] became a prominent paradigm for software development. It provides the means to design software systems by composing a number of components together to provide the required functionality. Since components are typically offered by different providers, the paradigm settles down a distributed environment for both intra- and inter-enterprise application integration and collaboration. Component orientation shifts the focus of software development towards component selection and composition.

Often, however, components are selected among behaviourally equivalent candidates, which differ in a number of non-functional characteristics. The latter define what is known as the system's 'Quality of Service' (QoS) properties. They include response time, availability, bandwidth requirement, memory usage, among many others, all of them playing a major role in systems design and expected to be propagated across composition. Moreover, often adaptation mechanisms have to take them into account, going far beyond the simple wrapping of functionality to bridge between published interfaces. The QoS designation is widely accepted to group together all these concerns [18, 27]. It suggests twin notions of a *level* to be attained and *cost* to be paid, as well as points out to the design of suitable metrics to quantify such properties. Dealing with QoS aspects in a coherent and systematic way became a main issue in component composition, which cannot be swept under the carpet in any formal account of the problem.

This paper extends a formal calculus for component composition, popularised under the slogan "*components as coalgebras*" [2, 22], in order to take into account, in an explicit way, QoS information. The calculus is based on a coalgebraic model used to capture components' observable behavior and persistence over transitions. Furthermore, it is parametric on a notion of *behavior*, encoded as a a strong monad [15], which allows to reason in a uniform way about components exhibiting different sorts of behaviour (*e.g.*, total or partial, non-deterministic or stochastic). In general, coalgebra theory [1, 20] nicely captures a "black-box" characterization of software components, which favors an observational, coinductive semantics: the essence of a component specification lies in the collection of *possible observations* and any two internal configurations should be identified wherever indistinguishable by observation.

QoS properties are typically analysed through quantitative stochastic methods (*e.g.*, queueing theory methods [8] or generalized stochastic Petri nets [17]) that explicitly take into account the impact of uncertainty on systems' design. Over the past few decades, these methods provided powerful ways to address and solve QoS problems in many application areas, such as telecommunication networks or software architecture. A bottleneck, however, of using quantitative stochastic methods to reason about end-to-end QoS properties of a complex system is the construction of a suitable model of the system in the first place. Traditional methods for construction of these models rely on the insight of experts. The resulting models often do not reflect the functional and/or architectural pieces used in the construction of the actual system. This makes them fragile, in the sense that even small changes to those functional/architectural building blocks may invalidate the model. Whereas these issues present problems in reasoning about QoS properties of traditional monolithic systems, they demand new approaches in the context of highly heterogeneous components offered by disparate providers, each of which guarantees particular QoS properties through their own service level contracts. In order to express QoS properties attached to each component and compute QoS levels across component composition, we adopt (a slight generalization) of the notion of Q-algebra proposed in [9]. Briefly, this consists of two constraint semirings over a common carrier, representing some form of a *cost* domain, which allows different ways of combining and choosing between quality values.

As the result, the calculus proposed in this paper provides a compositional approach in which complex components can be constructed by aggregation of more elementary ones while propagating the relevant QoS constraints. A preliminary version of this approach appeared in [3].

A major effect of introducing QoS-awareness is that a number of bisimilarity equations holding in the plain calculus [2, 5] become refinement laws, *i.e.*, inequalities with respect to some quality preorder. Proofs, however, can still be carried on in the *calculational* style which is the watermark of [2, 5, 22]. This style avoids the explicit construction of, *e.g.*, bisimulations, when proving observational equality, favoring an essentially point-free reasoning style, popularized, at the *micro* programming level, under the name of *algebra of programming* [7].

The remaining of this paper is devoted to substantiate this claim. As a second contribution, the paper introduces a prototyping tool for the QoS-aware component calculus, developed as a 'proof-of-concept' to model components and simulate their behaviour.

The paper structure is as follows. The original component calculus is summarised in the following section. Section 3 introduces its extension and redefines the basic operators to make components QoS aware. An illustrative example is discussed in section 4. Section 5 discusses what changes in the component calculus in the presence of QoS awareness and revisits a number of composition laws. The prototyping tool, developed in HASKELL, is discussed in section 6. Finally, section 7 concludes.

2. The plain calculus of software components

This section recalls the basic mechanisms for component aggregation along the lines of [2, 5, 22]. The interested reader will find in these references all the details and proofs omitted here.

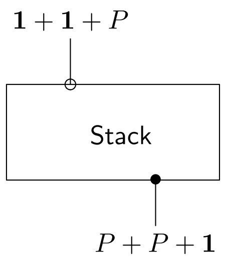
Under the slogan "*components as coalgebras*", software components are characterised as dynamic systems with a public interface and a private, encapsulated state. The relevance of state information precludes a 'process-like' (purely behavioral) view. Actually, components are *concrete* coalgebras [20, 1, 14]). For a given value of the state space — referred to as a *seed* in the sequel — a corresponding 'process', or *behavior*, arises by computing its coinductive extension. This coalgebraic modelling approach provides an observational semantics for software components and a generic assembly calculus.

A typical example of such a state-based component is the ubiquitous *stack*. Denoting by U its internal state, a stack of values of type P is handled through the usual

$$\mathsf{top} : \ U \longrightarrow P, \quad \mathsf{pop} : \ U \longrightarrow P \times U \quad \text{and} \quad \mathsf{push} : \ U \times P \longrightarrow U$$

operations. An alternative, 'black box' view hides U from the stack environment and regards each operation as a pair of input/output ports. For example, the top operation becomes declared as $\mathsf{top} : \ \mathbf{1} \longrightarrow P$, where $\mathbf{1}$ stands for the nullary (or unit) datatype. The intuition is that top is activated with the simple pushing of a 'button' (its argument being the stack private state space) whose effect is the production of a P value in the corresponding output port. Similarly typing push as $\mathsf{push} : \ P \longrightarrow \mathbf{1}$ means that an external argument is required on activation but no visible output is produced, but for a trivial indication of successful termination. Such 'port' signatures are grouped together in the diagram below. Combined input type $\mathbf{1} + \mathbf{1} + P$ models the choice of three functionalities (top, pop and push in this order), of which only one takes input of type P.

$$\begin{cases} \mathsf{pop}: & \mathbf{1} \longrightarrow P \\ \mathsf{top}: & \mathbf{1} \longrightarrow P \\ \mathsf{push}: & P \longrightarrow \mathbf{1} \end{cases}$$



Component Stack encapsulates a number of services through a public *interface* providing limited access to its internal *state space*. Furthermore, it *persists* and *evolves* in time, in a way which can only be traced through observations at the interface level. One might capture these intuitions by providing an explicit semantic definition in terms of a function $[\![\mathsf{Stack}]\!] : U \times I \longrightarrow (U \times O + \mathbf{1})$, where I, O abbreviate $\mathbf{1} + \mathbf{1} + P$ and $P + P + \mathbf{1}$, respectively. The presence of $\mathbf{1}$ in its result type indicates that the overall behaviour of this component is *partial*: in a number of state configurations the execution of some operations may fail. This function describes how Stack reacts to input stimuli, produces output data (if any) and changes state. It can also be written in a curried form as

$$\overline{[\![\mathsf{Stack}]\!]} : U \longrightarrow (U \times O + \mathbf{1})^I$$

that is, as a *coalgebra* $U \longrightarrow \mathsf{T}\, U$ for functor $\mathsf{T}\, X = ((X \times O) + \mathbf{1})^I$.

The Stack example illustrates the basic elements of a semantic model for state-based components: *a)* the presence of an *internal state space* which evolves and persists in time, and *b)* the possibility of *interaction* with other components through well-defined interfaces and during the overall computation. This favours adoption of a *coalgebraic* modelling framework: components are inherently dynamic, possess an observable behaviour, but their internal configurations remain hidden and should be identified if not distinguishable by observation. Qualifier 'state-based' is used in the sense the word 'state' has in automata theory — the internal memory of the automaton which both constrains and is constrained by the execution of component operations. Such operations are encoded in a functor which constitutes the (syntax of the) component interface.

In the simplest, deterministic case, the behavior of a component p is captured by the output it produces, which is determined by the supplied input and the current internal state . But reality is often more complicated, for one may have to deal with components whose behavioral pattern is, e.g., partial or even non-deterministic. Therefore, it is helpful to proceed in a generic way. *Genericity* means that the proposed constructions are parametric on a (mathematical) model of behavior. Genericity is achieved by abstracting a given behavior model by an arbitrary *strong monad* B. Recall that a *strong monad* is a monad $\langle \mathsf{B}, \eta, \mu \rangle$ where B is a strong functor and both η and μ are strong natural transformations [15]. B being strong means there exist natural transformations $\tau_r^{\mathsf{T}} : \mathsf{T} \times - \Longrightarrow \mathsf{T}(\mathsf{Id} \times -)$ and $\tau_l^{\mathsf{T}} : - \times \mathsf{T} \Longrightarrow \mathsf{T}(- \times \mathsf{Id})$, called the right and left strength, respectively, subject to certain conditions. Their effect is to *distribute* the free variable values in the context "$-$" along functor B. The Kleisli composition of the right with the left strength, gives rise to a natural transformation whose component on objects I and J is given by $\delta_r = \tau_{r_{I,J}} \bullet \tau_{l_{\mathsf{B}I,J}}$ Dually, $\delta_l = \tau_{l_{I,J}} \bullet \tau_{r_{I,\mathsf{B}J}}$. Such transformations specify how the monad distributes over product and, therefore, represent a sort of sequential composition of B-computations.

For example, $\mathsf{B} = \mathsf{Id}$ retrieves the simple deterministic behavior, whereas $\mathsf{B} = \mathcal{P}$ or $\mathsf{B} = \mathsf{Id}+\mathbf{1}$ would model non-deterministic or partial behavior, respectively. Recall $\mathcal{P}$ is the finite powerset monad. On the other hand, $\mathbf{1}$ represents abstractly a singleton set; therefore type $X+\mathbf{1}$ means *either X or undefined*. The notation used in the sequel is quite standard in mathematics for computer science; reference [7] provides an excellent introduction. Assume a collection of sets I, O, ..., acting as component interfaces, *i.e.*, input and output range of components. Then this abstraction leads to coalgebras for functor

$$\mathsf{T}^{\mathsf{B}} = (\mathsf{B}(\mathsf{Id} \times O))^I \tag{2.1}$$

as a possible general model for state based software components. Formally,

Definition 2.1. A component $p : I \longrightarrow O$ taking input in I and producing output in O is specified by a pointed coalgebra

$$\langle u_p \in U_p, \overline{a}_p : U_p \longrightarrow (\mathsf{B}(U_p \times O))^I \rangle \tag{2.2}$$

where u_p is the initial state, often referred to as the *seed*, and the coalgebra dynamics is captured by currying a state-transition function $a_p : U_p \times I \longrightarrow \mathsf{B}\ (U_p \times O)$.

This definition means that the computation of an action in a component will not simply produce an output and a continuation state, but a B-structure of such pairs. The monadic structure provides tools to handle such computations. Unit (η) and multiplication (μ), provide, respectively, a value embedding and a 'flatten' operation to reduce nested behavioral effects. Strength, either in its right (τ_r) or left (τ_l) version, cater for context information.

Having defined generic components as (pointed) coalgebras, one may wonder how do they get composed and what kind of calculus emerges from this framework. In this framework, interfaces are sets representing the input and output range of a component. Consequently, components are arrows between interfaces and so arrows between components are arrows between arrows. Thus, three notions have to be taken into account: interfaces, components and component morphisms. Formally, this leads to a *bicategorial* setting, but we will avoid such an abstraction step in the sequel. For the moment retain that a component morphism $h : \langle u_p, \overline{a}_p \rangle \longrightarrow \langle u_q, \overline{a}_q \rangle$ is just a function connecting the state spaces of p and q and satisfying the following *morphism* and *seed preservation* conditions:

$$\overline{a}_q \cdot h = \mathsf{T}^{\mathsf{B}}\ h \cdot \overline{a}_p \tag{2.3}$$

$$h\ u_p = u_q \tag{2.4}$$

Components with compatible interfaces (for example, $p : I \longrightarrow K$ and $q : K \longrightarrow O$) can be composed sequentially as follows:

$$p\,;q = \langle \langle u_p, u_q \rangle \in U_p \times U_q, \overline{a}_{p;q} \rangle$$

where $a_{p;q} : U_p \times U_q \times I \longrightarrow \mathsf{B}(U_p \times U_q \times O)$ is detailed as follows

$$\begin{aligned} a_{p;q} &= U_p \times U_q \times I \xrightarrow{\mathsf{xr}} U_p \times I \times U_q \xrightarrow{a_p \times \mathsf{id}} \\ &\mathsf{B}(U_p \times K) \times U_q \xrightarrow{\tau_r} \mathsf{B}(U_p \times K \times U_q) \xrightarrow{\mathsf{B}(\mathsf{a} \cdot \mathsf{xr})} \\ &\mathsf{B}(U_p \times (U_q \times K)) \xrightarrow{\mathsf{B}(\mathsf{id} \times a_q)} \mathsf{B}(U_p \times \mathsf{B}(U_q \times O)) \\ &\xrightarrow{\mathsf{B}\tau_l} \mathsf{BB}(U_p \times (U_q \times O)) \xrightarrow{\mathsf{BBa}^\circ} \\ &\mathsf{BB}(U_p \times U_q \times O) \xrightarrow{\mu} \mathsf{B}(U_p \times U_q \times O) \end{aligned}$$

The identity of sequential composition is component $\mathsf{copy}_K : K \longrightarrow K$, where

$$\mathsf{copy}_K = \langle * \in \mathbf{1}, \overline{a}_{\mathsf{copy}_K} \rangle$$

and $a_{\mathsf{copy}_K} = \eta$.

The definition above resorts to common 'housekeeping' morphisms such as product and sum associativity, (a, a_+), commutativity (s, s_+), left and right units (l, l_+ and r, r_+), left and right distributivity (dl, dr) and isomorphisms $\mathsf{xl} : A \times (B \times C) \longrightarrow B \times (A \times C)$, $\mathsf{xr} : A \times B \times C \longrightarrow A \times C \times B$ and $\mathsf{m} : (A \times B) \times (C \times D) \longrightarrow (A \times C) \times (B \times D)$. Note that, by convention, binary morphisms always associate to the left. As one would expect, reasoning about generic components entails a number of laws relating monads with this sort of morphisms. Such laws are thoroughly dealt with in [5].

Recall (from *e.g.* [20]) that the graph of a morphism is a bisimulation. Therefore, the existence of a seed preserving morphism between two components makes them T^{B}-bisimilar, leading to the following laws, for appropriately typed components p, q and r:

$$\mathsf{copy}_I \,;\, p \sim p \sim p \,;\, \mathsf{copy}_O \tag{2.5}$$

$$(p \,;\, q) \,;\, r \sim p \,;\, (q \,;\, r) \tag{2.6}$$

In [2] a collection of component combinators was defined and their properties were studied. The component calculus starts by showing that any function $f : A \longrightarrow B$ can be lifted to a component whose interfaces are given by their domain and codomain types. Formally, a function $f : A \longrightarrow B$ gives rise to component

$$\ulcorner f \urcorner = \langle * \in \mathbf{1}, \overline{a}_{\ulcorner f \urcorner} \rangle$$

i.e., a coalgebra over $\mathbf{1}$ whose action is given by the currying of

$$a_{\ulcorner f \urcorner} = \mathbf{1} \times A \xrightarrow{\mathsf{id} \times f} \mathbf{1} \times B \xrightarrow{\eta_{(\mathbf{1} \times B)}} \mathsf{B}(\mathbf{1} \times B) \tag{2.7}$$

A *wrapping* mechanism $p[f, g]$ which encodes the pre- and post-composition of a component with functions is defined as a combinator which resembles the renaming connective found in process algebras (*e.g.*, [19]). Let $p : I \longrightarrow O$ be a component and consider functions $f : I' \longrightarrow I$ and $g : O \longrightarrow O'$. Component p wrapped by f and g, denoted by $p[f, g]$ and typed as $I' \longrightarrow O'$, is defined by input pre-composition with f and output post-composition with g. Formally, it maps component p from $\langle u_p, \overline{a}_p \rangle$ into $\langle u_p, \overline{a}_{p[f,g]} \rangle$, where

$$a_{p[f,g]} = U_p \times I' \xrightarrow{\mathsf{id} \times f} U_p \times I \xrightarrow{a_p} \mathsf{B}(U_p \times O) \xrightarrow{\mathsf{B}(\mathsf{id} \times g)} \mathsf{B}(U_p \times O')$$

Parallel composition, denoted by $p \boxtimes q$, corresponds to a synchronous product: both components are executed simultaneously when triggered by a pair of legal input values. Note, however, that the behavior effect, captured by monad B, propagates. For example, if B can express component failure and one of the arguments fails, product fails as well. Formally,

$$p \boxtimes q \;=\; \langle \langle u_p, u_q \rangle \in U_p \times U_q, \overline{a}_{p\boxtimes q} \rangle$$

where

$$\begin{aligned} a_{p\boxtimes q} &= U_p \times U_q \times (I \times J) \xrightarrow{\mathsf{m}} U_p \times I \times (U_q \times J) \\ &\xrightarrow{a_p \times a_q} \mathsf{B}\,(U_p \times O) \times \mathsf{B}\,(U_q \times R) \xrightarrow{\delta_l} \\ &\mathsf{B}\,(U_p \times O \times (U_q \times R)) \xrightarrow{\mathsf{B\ m}} \mathsf{B}\,(U_p \times U_q \times (O \times R)) \end{aligned}$$

and maps every pair of arrows $\langle h_1, h_2 \rangle$ into $h_1 \times h_2$.

Dual to parallel composition is *external choice* denoted in the calculus by $\boxplus$. When interacting with $p \boxplus q : I + J \to O + R$, the environment chooses either to input a value of type I or one of type J, which triggers the corresponding component (p or q, respectively), producing the relevant output. Formally, the *choice* combinator is defined as a lax functor $\boxplus : \mathsf{Cp} \times \mathsf{Cp} \longrightarrow \mathsf{Cp}$, which consists of an action on objects given by $I \boxplus J \;=\; I + J$ and a family of functors

$$\boxplus_{I,O,J,R} : \mathsf{Cp}(I, O) \times \mathsf{Cp}(J, R) \longrightarrow \mathsf{Cp}(I + J, O + R)$$

yielding

$$p \boxplus q \;=\; \langle \langle u_p, u_q \rangle \in U_p \times U_q, \overline{a}_{p\boxplus q} \rangle$$

$$\begin{aligned} a_{p\boxplus q} \;=\; & U_p \times U_q \times (I + J) \xrightarrow{(\mathsf{xr}+\mathsf{a})\cdot\mathsf{dr}} U_p \times I \times U_q + U_p \times (U_q \times J) \\ & \xrightarrow{a_p \times \mathsf{id} + \mathsf{id} \times a_q} \mathsf{B}\,(U_p \times O) \times U_q + U_p \times \mathsf{B}\,(U_q \times R) \\ & \xrightarrow{\tau_r + \tau_l} \mathsf{B}\,(U_p \times O \times U_q) + \mathsf{B}\,(U_p \times (U_q \times R)) \\ & \xrightarrow{\mathsf{Bxr}+\mathsf{Ba}^\circ} \mathsf{B}\,(U_p \times U_q \times O) + \mathsf{B}\,(U_p \times U_q \times R) \\ & \xrightarrow{[\mathsf{B}\ (\mathsf{id}\times\iota_1), \mathsf{B}\ (\mathsf{id}\times\iota_2)]} \mathsf{B}\,(U_p \times U_q \times (O + R)) \end{aligned}$$

and mapping pairs of arrows $\langle h_1, h_2 \rangle$ into $h_1 \times h_2$.

Still another tensor, $p \circledast q$, expresses *concurrent composition* by combining choice and parallel, in the sense that p and q can be executed independently or jointly, depending on the input supplied.

Finally, generalized interaction is catered through a sort of "feedback" mechanism on a subset of the inputs. This is also defined by a combinator, called *hook*, which connects some input to some output wires and, consequently, forces part of the output of a component to be fed back as input. Formally, the *hook* combinator $- \urcorner_Z$ maps each component $p : I + Z \longrightarrow O + Z$ to $p \urcorner_Z : I + Z \longrightarrow O + Z$ given by

$$p \urcorner_Z = \; \langle u_p \in U_p, \overline{a}_{p\urcorner_Z} \rangle$$

where

$$\begin{aligned} a_{p\Lsh_Z} &= U_p \times (I+Z) \xrightarrow{a_p} \mathsf{B}(U_p \times (O+Z)) \\ &\xrightarrow{\mathsf{B}((\mathsf{id}\times\iota_1+\mathsf{id}\times\iota_2)\cdot\mathsf{dr})} \mathsf{B}(U_p \times (O+Z) + U_p \times (I+Z)) \\ &\xrightarrow{\mathsf{B}(\eta+a_p)} \mathsf{B}(\mathsf{B}(U_p \times (O+Z)) + \mathsf{B}(U_p \times (O+Z))) \\ &\xrightarrow{\mu\cdot\mathsf{B}\nabla} \mathsf{B}(U_p \times (O+Z)) \end{aligned}$$

3. QoS-aware components

Modelling QoS

QoS is introduced in the calculus of (plain) components through (a slight generalisation of) the notion of a Q-algebra due to [9].

Definition 3.1. Given a set C of QoS values, a Q-algebra is a structure $(C, \oplus, \otimes, \obar, \mathbf{0}, \mathbf{1})$ such that $R_\otimes = (C, \oplus, \otimes, \mathbf{0}, \mathbf{1})$ and $R_\obar = (C, \oplus, \obar, \mathbf{0}, \mathbf{1})$ are both constraint semirings over C.

Intuitively, C is a QoS domain (*e.g.*, a measure of resource usage or availability) whereas $\oplus$ represents a *choice* between QoS values and $\otimes$ and $\obar$, respectively, compose them sequentially and concurrently. The former act as an *addition*, to select among costs; the latter two as two forms of *multiplication* to combine them. Given two costs, $\oplus$ returns their least upper bound, while $\otimes$ and $\obar$ compute an aggregated cost. The identity $\mathbf{0}$ of $\oplus$ is the least possible cost value, whereas element $\mathbf{1}$, the common identity of $\otimes$ and $\obar$ represent a neutral cost value. Combining costs concurrently or sequentially will not affect this neutral cost element so the identity is common to both *multiplications*.

The definition of a Q-algebra entails distribution of both $\otimes$ and $\obar$ over $\oplus$, and defines the identity of $\oplus$ as an absorbing (zero) element for both *multiplications*:

$$(a \oplus b) \circ c = (a \circ c) \oplus (b \circ c) \tag{3.1}$$

$$c \circ (a \oplus b) = (c \circ a) \oplus (c \circ b) \tag{3.2}$$

$$\mathbf{0} \circ a = \mathbf{0} \tag{3.3}$$

where $\circ = \otimes, \obar$. In a constraint semiring $\oplus$ is idempotent, therefore entailing the definition of a partial order

$$a \geq b \Leftrightarrow (a \oplus b) = b \tag{3.4}$$

meaning a has a *higher cost* than b, or, equivalently, a lower QoS level. The following example helps to build up intuition about this structure.

Example 1. An example of a Q-algebra is the structure $(\mathbb{R}_+ \cup \{\infty\}, min, +, max, \infty, 0)$, where the QoS values are non-negative real numbers together with infinity, which can be used to specify the (execution) time. The additive operation is min, the sequential combinator is sum, and the concurrent combinator is max over the domain. Furthermore, the $\mathbf{0}$ and $\mathbf{1}$ in a Q-algebra structure correspond to ∞ and 0 respectively. This structure formalizes the notion that, given two components X and Y, (1) if X and Y are functionally equivalent, the execution time of X or Y is selected by applying the min operation to the time values of X and Y; (2) the execution time of a component resulting from the sequential composition of X and Y is obtained by applying the sum operation to the time values of X and Y; and (3) the execution time of a component resulting from the parallel composition of X and Y is obtained by applying the max operation to the execution time values of X and Y.

With respect to [9], we additionally require both $\otimes$ and $\oplus$ to be commutative, which makes easier formal manipulation in proofs. More fundamentally, we also require that the common identity of these cost combinators acts as an absorving element wrt $\oplus$, *i.e.*

$$\mathbf{1} \oplus a \;=\; \mathbf{1} \tag{3.5}$$

Law (3.5) is required to establish essential properties relating the QoS levels of individual components with their composition. In particular, we have

Lemma 3.1.

$$a \circ b \geq a \;\; \textit{and} \;\; a \circ b \geq b \qquad \textit{for} \circ = \otimes, \oplus \tag{3.6}$$

Proof.

$$\begin{array}{ll} & a \circ b \geq a \\ \Leftrightarrow & \quad \{ \text{(3.4)} \} \\ & (a \circ b) \oplus a = a \\ \Leftrightarrow & \quad \{ \text{identity for} \circ \} \\ & (a \circ b) \oplus (a \circ \mathbf{1}) = a \\ \Leftrightarrow & \quad \{ \text{(3.2)} \} \\ & a \circ (b \oplus \mathbf{1}) = a \\ \Leftrightarrow & \quad \{ \text{(3.5)} \} \\ & a \circ \mathbf{1} = a \\ \Leftrightarrow & \quad \{ \text{identity for} \circ \} \\ & a = a \end{array}$$

□

Concrete examples of (generalised) Q-algebras are given in section 4. Further examples of related structures are discussed, although in a different context, in [21].

QoS-aware Components

QoS information is included in the component model as an additional attribute: its execution generates a QoS value which is observable (*i.e.*, measurable). Formally, definition (2.1) changes to

Definition 3.2. A cost component $p : I \longrightarrow O$ is specified by a pointed coalgebra

$$\langle u_0 \in U_p, \overline{a}_p : U_p \to \mathsf{B}(U_p \times (C \times O))^I \rangle \tag{3.7}$$

where C is the domain of some Q-algebra $(C, \oplus, \otimes, \oslash, \mathbf{0}, \mathbf{1})$. Again u_p is the initial state and the coalgebra dynamics is captured by currying a state-transition function $a_p : U_p \times I \longrightarrow \mathsf{B}\ (U_p \times (C \times O))$.

The specification of all component combinators change accordingly to take into account the need for combining the observed QoS levels of their parameters. The following definitions give the *cost*-version of the coalgebra dynamics for all combinators mentioned in section 2.

As discussed above, the basic mechanism provided in the calculus to adapt component interfaces is *wrapping*. It consists of a component pre and post composition with suitable functions which modify the input and output universes without interfering with the component dynamics. Formally, in the presence of QoS information abstracted on a data parameter C, the dynamics of component $p : I \longrightarrow O$ wrapped by functions $f : I' \longrightarrow I$ and $g : O \longrightarrow O'$, denoted by $p[f, g]$, is defined as follows:

Definition 3.3 (wrapping with costs).

$$\begin{aligned} a_{p[f,g]} &= U_p \times I' \xrightarrow{\mathsf{id} \times f} U_p \times I \xrightarrow{a_p} \mathsf{B}(U_p \times (C \times O)) \\ &\xrightarrow{\mathsf{B}(\mathsf{id} \times (\mathsf{id} \times g))} \mathsf{B}(U_p \times (C \times O')) \end{aligned}$$

From this definition it is clear that the QoS level of $p[f, g]$ is that of p.

The calculus also allows to regard functions as particular instances of components, whose interfaces are given by their domain and codomain types. As discussed in section 2, a function $f : A \longrightarrow B$ is represented by component $\ulcorner f \urcorner$. Assigning a cost (or generically a QoS measure) $c \in C$ to the execution of f corresponds to changing (2.7) to

Definition 3.4 (function lifting with costs).

$$a_{\ulcorner f \urcorner} = \mathbf{1} \times A \xrightarrow{\mathsf{id} \times \langle \underline{c}, f \rangle} \mathbf{1} \times (C \times B) \xrightarrow{\eta} \mathsf{B}(\mathbf{1} \times (C \times B))$$

Components are put together through a number of tensors encoding sequential composition (either total or partial, the latter in the form of a *feedback* mechanism), parallel composition and choice. In the sequel the *cost*-versions of these combinators are given.

Definition 3.5 (sequential composition with costs).

$$\begin{aligned} a_{p;q} &= U_p \times U_q \times I \xrightarrow{\mathsf{xr}} U_p \times I \times U_q \xrightarrow{a_p \times \mathsf{id}} \mathsf{B}(U_p \times (C \times K)) \times U_q \\ &\xrightarrow{\tau_r} \mathsf{B}(U_p \times (C \times K) \times U_q) \xrightarrow{\mathsf{Bxr}} \mathsf{B}(U_p \times U_q \times (C \times K)) \\ &\xrightarrow{\mathsf{Bm}} \mathsf{B}(U_p \times C \times (U_q \times K)) \\ &\xrightarrow{\mathsf{B}(\mathsf{id} \times a_q)} \mathsf{B}(U_p \times C \times \mathsf{B}(U_q \times (C \times O))) \\ &\xrightarrow{\mathsf{B}\tau_l} \mathsf{BB}(U_p \times C \times (U_q \times (C \times O))) \\ &\xrightarrow{\mu} \mathsf{B}(U_p \times C \times (U_q \times (C \times O))) \\ &\xrightarrow{\mathsf{Bm}} \mathsf{B}(U_p \times U_q \times (C \times (C \times O))) \\ &\xrightarrow{\mathsf{B}(\mathsf{id} \times \mathsf{a}^\circ)} \mathsf{B}(U_p \times U_q \times ((C \times C) \times O)) \end{aligned}$$

$$\xrightarrow{\mathsf{B}(\mathsf{id}\times(\otimes\times\mathsf{id}))} \mathsf{B}(U_p \times U_q \times (C \times O))$$

Note the use of $\otimes$ to sequentially compose QoS levels. The same occurs in the redefinition of the *hook* combinator

$$p \urcorner_Z = \langle u_p \in U_p, \overline{a}_{p\urcorner_Z} : U_p \to \mathsf{B}(U_p \times C \times (O+Z))^{I+Z} \rangle$$

which is essentially a generalization of sequential composition:

Definition 3.6 (hook with costs).

$$\begin{aligned} a_{p\urcorner_Z} &= U_p \times (I+Z) \xrightarrow{a_p} \mathsf{B}(U_p \times (C \times (O+Z))) \xrightarrow{\mathsf{B}((\mathsf{dr}\times\mathsf{id})\cdot\mathsf{a}^\circ\cdot(\mathsf{id}\times\mathsf{s}))} \\ &\mathsf{B}((U_p \times O + U_p \times Z) \times C) \xrightarrow{\mathsf{B}((\mathsf{id}\times\iota_1+\mathsf{id}\times\iota_2)\times\mathsf{id})} \\ &\mathsf{B}(((U_p \times (O+Z) + U_p \times (I+Z)) \times C) \xrightarrow{\mathsf{B}(((\eta\cdot(\mathsf{id}\times\langle\underline{1}\cdot!,\mathsf{id}\rangle)+a_p)\times\mathsf{id})} \\ &\mathsf{B}((\mathsf{B}(U_p \times (C \times (O+Z))) + \mathsf{B}(U_p \times (C \times (O+Z)))) \times C) \xrightarrow{\mathsf{B}(\triangledown\times\mathsf{id})} \\ &\mathsf{B}(\mathsf{B}(U_p \times (C \times (O+Z))) \times C) \xrightarrow{\mu\cdot\mathsf{B}(\mathsf{B}((\mathsf{id}\times(\otimes\times\mathsf{id}))\cdot(\mathsf{id}\times\mathsf{xr})\cdot\mathsf{a})\cdot\tau_r)} \\ &\mathsf{B}(U_p \times (C \times (O+Z))) \end{aligned}$$

Recalling that $\mathbf{1}$ is the identity of $\otimes$, observe how term $\eta\cdot(\mathsf{id}\times\langle\underline{\mathbf{1}}\cdot!,\mathsf{id}\rangle)$ induces a neutral cost in the expression branch where no further execution of a_p occurs. To clarify notation, note that function ! is the universal arrow to the singleton set yielding composition

$$O+Z \xrightarrow{!} \mathbf{1} \xrightarrow{\underline{1}} C$$

On its turn, function $\triangledown : X+X \longrightarrow X$ is the codiagonal defined by $\triangledown = [\mathsf{id},\mathsf{id}]$.

The redefinition of parallel composition, on its turn, resorts to $\oplus$, as follows:

Definition 3.7 (parallel composition with costs).

$$\begin{aligned} a_{p\boxtimes q} &= U_p \times U_q \times (I \times J) \xrightarrow{\mathsf{m}} (U_p \times I) \times (U_q \times J) \\ &\xrightarrow{a_p\times a_q} \mathsf{B}(U_p \times (C \times O)) \times \mathsf{B}(U_q \times (C \times K)) \\ &\xrightarrow{\delta_l} \mathsf{B}((U_p \times (C \times O)) \times (U_q \times (C \times K))) \\ &\xrightarrow{\mathsf{Bm}} \mathsf{B}(U_p \times U_q \times ((C \times O) \times (C \times K))) \\ &\xrightarrow{\mathsf{B}(\mathsf{id}\times\mathsf{m})} \mathsf{B}(U_p \times U_q \times ((C \times C) \times (O \times K))) \\ &\xrightarrow{\mathsf{B}(\mathsf{id}\times(\oplus\times\mathsf{id}))} \mathsf{B}(U_p \times U_q \times C \times (O \times K)) \end{aligned}$$

Finally, component *choice* becomes

Definition 3.8 (choice with costs).

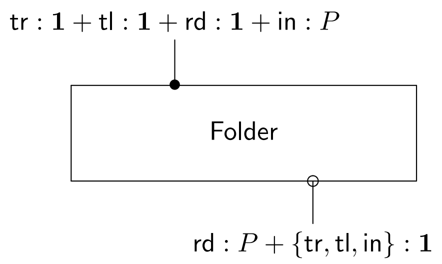
$$\begin{aligned} a_{p\boxplus q} &= U_p \times U_q \times (I \times J) \xrightarrow{(\mathsf{xr}+a)\cdot\mathsf{dr}} U_p \times I \times U_q + U_p \times (U_q \times J) \\ &\xrightarrow{a_p\times\mathsf{id}+\mathsf{id}\times a_q} \mathsf{B}(U_p \times (C \times O)) \times U_q + U_p \times \mathsf{B}(U_q \times (C \times R)) \end{aligned}$$

$$\xrightarrow{\tau_r+\tau_l} \mathsf{B}(U_p \times (C \times O) \times U_q) + \mathsf{B}(U_p \times U_q \times (C \times R)))$$
$$\xrightarrow{\mathsf{B}(\mathsf{xr}+\mathsf{id})} \mathsf{B}(U_p \times U_q \times (C \times O)) + \mathsf{B}(U_p \times U_q \times (C \times R))$$
$$\xrightarrow{[\mathsf{B}(\mathsf{id}\times\iota_1),\mathsf{B}(\mathsf{id}\times\iota_2)]} \mathsf{B}(U_p \times U_q \times (C \times O + C \times R))$$
$$\xrightarrow{\mathsf{B}(\mathsf{id}\times\mathsf{dr}^\circ)} \mathsf{B}(U_p \times U_q \times (C \times (O + R)))$$

4. Example: A folder from two stacks

The purpose of this section is to illustrate how new components can be built from old ones, relying solely on the functionality available. The example is the construction of a *folder* out of two *stacks*. Although these components are parametric on the type of stacked objects, we will refer to these as 'pages', by analogy with a folder in which new 'pages' are inserted on and retrieved ('read') from the righthandside pile.

The specification, the Folder component should provide operations to *read*, *insert* a new page, *turn a page right* and *turn a page left*. Reading returns the page which is immediately accessible once the folder is open at some position. Insertion takes as argument the page to be inserted. The other two operations are simply state updates. Let P be the type of a *page*. The Folder 'port' signature may be represented as follows, where input and output types are decorated with the corresponding action names:



Our exercise consists in building Folder assuming that two stacks are used to model the *left* and *right* piles of pages, respectively. The intuition is that the push action of the right stack will be used to model page insertion into the folder, *i.e.*, action in. On the other hand, it should also be connected to the pop of the left one to model tr, the 'turn page right' action. A symmetric connection will be used to model tl. The rd operation observes the 'front' page — the one which can be accessed by top on the right stack.

According to this plan, the assembly of Folder starts by defining RightS as a Stack component suitably wrapped to meet the above mentioned constraints. At the input level we need to replicate the input to push by wrapping p with the codiagonal $\triangledown_P$ function. On the other hand, access to the top button on the left stack is removed by ι_2. At the output level, because of the additive interface structure, we cannot get rid of the top result. It is possible, however, to associate it to the push output and collapse both into $\mathbf{1}$, via $!_{P+\mathbf{1}}$. So we define:

$$\mathsf{RightS} \;=\; \mathsf{Stack}[\mathsf{id} + \triangledown, \mathsf{id}] \;:\; \mathbf{1} + \mathbf{1} + (P + P) \longrightarrow P + P + \mathbf{1}$$

$$\mathsf{LeftS} \;=\; \mathsf{Stack}[\iota_2 + \mathsf{id}, (\mathsf{id}+!_{P+\mathbf{1}}) \cdot \mathsf{a}_+] \;:\; \mathbf{1} + P \longrightarrow P + \mathbf{1}$$

Then, we form the $\boxplus$ composition of both components:

$$\mathsf{LeftS} \boxplus \mathsf{RightS} : \; \mathbf{1} + P + (\mathbf{1} + \mathbf{1} + (P + P)) \longrightarrow P + \mathbf{1} + (P + P + \mathbf{1})$$

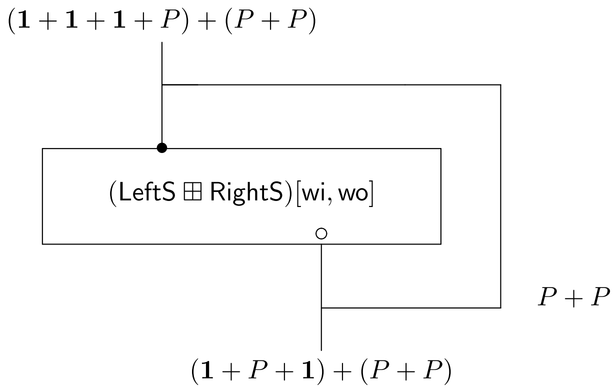
The next step builds the desirable connections using *hook* over this composite, which requires a previous wrapping by a pair of suitable isomorphisms:

$$((\mathsf{LeftS} \boxplus \mathsf{RightS})[\mathsf{wi}, \mathsf{wo}]) \,\Lsh_{P+P}$$

where, denoting by ι_{ij} the composite $\iota_i \cdot \iota_j$,

$$\mathsf{wi} \;=\; \Big[[[[\iota_{11}, \iota_{211}], \iota_{212}], \iota_{222}], [\iota_{221}, \iota_{12}]\Big] \qquad \mathsf{wo} \;=\; \Big[[\iota_{21}, \iota_{111}], [[\iota_{22}, \iota_{112}], \iota_{12}]\Big]$$

In a diagram:


$(\mathbf{1} + \mathbf{1} + \mathbf{1} + P) + (P + P)$
$(\mathsf{LeftS} \boxplus \mathsf{RightS})[\mathsf{wi}, \mathsf{wo}]$
$P + P$
$(\mathbf{1} + P + \mathbf{1}) + (P + P)$

Finally, to conform this component to the Folder interface, we restrict the feed back input — by pre-composing with $\mathsf{fi} = \iota_1$ — and collapse both the trivial output and the feed back one to $\mathbf{1}$, by post-composing with $\mathsf{fo} = \big[[[\iota_2, \iota_1], \iota_2], \iota_2 \cdot !_{P+P}\big]$. Therefore, we complete the exercise by defining

$$\mathsf{Folder} \;=\; (((\mathsf{LeftS} \boxplus \mathsf{RightS})[\mathsf{wi}, \mathsf{wo}]) \,\Lsh_{P+P})[\mathsf{fi}, \mathsf{fo}]$$

which respects the intended interface. Note this design retains the *architecture* of the 'folder' component without any commitment to a particular behaviour model.

Consider, now, two components satisfying the Stack interface. One of them, PlainStack, implements the three operations (push, pop and top) over a sequence of elements acting as its internal state space. The other one, CostStack, has a cost associated to pushing a new element onto the stack composed of two dimensions: reliability and time. These two quality dimensions are represented by the corresponding Q-algebras

$$\begin{aligned} Q_1 &= ([0,1], max, \star, \star, 0, 1) \\ Q_2 &= (\mathbb{R}_+ \cup \{\infty\}, min, +, max, \infty, 0) \end{aligned}$$

Furthermore, they can be composed to yield a new Q-algebra for the two QoS dimensions. Elements of this composed structure are pairs of elements of Q_1 and Q_2. Formally

$$Q \;= ([0,1] \times (\mathbb{R}_+ \cup \{\infty\}, max \times min, \star \times +, \star \times max, (0,\infty), (1,0))$$

Let $C = [0,1] \times (\mathbb{R}_+ \cup \{\infty\})$ be the domain of Q, and $T = \mathbb{R}_+ \cup \{\infty\}$ be the set of time values. The three operations of CostStack are specified as follows:

$$\begin{cases} \mathsf{push}(u,p) \;=\; \iota_2\,(p : u, ((c_r, c_t), *)) \\ \mathsf{pop}(u,*) \;=\; \begin{cases} \iota_2 * & \text{if } \; u = [] \\ \iota_1(\mathsf{tl}\, u, ((1,0), \mathsf{hd}\, u)) & \text{if } \; u \neq [] \end{cases} \\ \mathsf{top}(u,*) \;=\; \begin{cases} \iota_2 * & \text{if } \; u = [] \\ \iota_1(u, ((1,0), \mathsf{hd}\, u)) & \text{if } \; u \neq [] \end{cases} \end{cases}$$

where $c_r \in [0,1]$ and $c_t \in \mathbb{R}_+ \cup \{\infty\}$, are, respectively, a reliably cost and the time of pushing an element onto the stack. Note how the other two operations are performed at neutral costs (for the sequential and parallel composition). In the case of PlainStack all three operations have just neutral costs. Clearly, implementing the Folder component based on PlainStack or CostStack, or a combination thereof, yields components with similar behaviour but different QoS levels

5. Revisiting composition laws

5.1. A refinement relation

The purpose of this section is to revisit the component calculus introduced in [2, 5], and briefly summarised in section 2, in the presence of explicit QoS constraints formalised through a Q-algebra. The fundamental observation is that most laws stated as bisimilarity equations in the plain calculus, become inequalities with respect to some order capturing the increase of QoS levels. In the terminology of [22] the calculus becomes essentially a *refinement* calculuss. This is not unexpected. Actually, note that a bisimulation over functor $(\mathsf{B}(X \times (C \times O)))^I$, entails strict equality at the interface level (values of types I, O and C) and, thus, equal costs.

Therefore, to compare component-based designs with respect to QoS measures entails the need for a notion of *refinement*: $q \trianglelefteq p$ (read 'component q refines p') if they exhibit the same behavior but the QoS level of q is higher (*i.e.*, 'costs less') than that of p. Formally, this is recorded by the existence of what was called in [22] a *forward* morphism with respect to a preorder which captures a cost reduction (or, equivalently, an increase in QoS levels).

We will proceed in a generic way, defining such a preorder $\sqsubseteq_{\mathsf{T}}$ for each regular functor T. Then, definition 5.2 will characterise the refinement order relevant for reasoning about QoS-aware components.

Recall first that a T-coalgebra morphism $h : \alpha \longrightarrow \beta$ is a function from the state space of α to that of β such that

$$\mathsf{T}h \cdot \alpha \; = \; \beta \cdot h \tag{5.1}$$

It is well known that the existence of such a morphism entails bisimulation. A weak form of morphism where (5.1) is rephrased in terms of a preorder. $\sqsubseteq_{\mathsf{T}}$ over functor T was proposed in [22], where it is called a *forward* morphism:

$$\mathsf{T}h \cdot \alpha \; \dot{\sqsubseteq}_{\mathsf{T}} \; \beta \cdot h \tag{5.2}$$

Note that we denote by $\dot{\sqsubseteq}_{\mathsf{T}}$ the pointwise lifting of $\sqsubseteq_{\mathsf{T}}$ to the functional level,

$$f \; \dot{\sqsubseteq}_{\mathsf{T}} \; g \; \Leftrightarrow \; \forall_x . \; f \; x \sqsubseteq_{\mathsf{T}} g \; x$$

which can also be formulated in the following pointfree way,

$$f \; \dot{\sqsubseteq}_{\mathsf{T}} \; g \; \Leftrightarrow \; (f \subseteq \sqsubseteq_{\mathsf{T}} \cdot g)$$

Reference [22] proves that such a morphism still preserves transitions, therefore entailing a form of *simulation*, and that, for a given functor T, coalgebras and forward morphisms form a category.
Let us then define a preorder $\sqsubseteq_{\mathsf{T}}$, to capture cost reduction as follows:

Definition 5.1. For a regular functor T, the *cost reduction preorder* $\sqsubseteq_{\mathsf{T}}$ is defined by

$$\begin{array}{rll}
x \sqsubseteq_{\mathsf{Id}} y & \text{iff} & x = y \\
x \sqsubseteq_K y & \text{iff} & \begin{cases} K = C & \Rightarrow x \leq y \\ K \neq C & \Rightarrow x =_K y \end{cases} \\
x \sqsubseteq_{\mathsf{T}_1 \times \mathsf{T}_2} y & \text{iff} & \pi_1 \; x \sqsubseteq_{\mathsf{T}_1} \pi_1 \; y \; \wedge \; \pi_2 \; x \sqsubseteq_{\mathsf{T}_2} \pi_2 \; y \\
x \sqsubseteq_{\mathsf{T}_1 + \mathsf{T}_2} y & \text{iff} & \begin{cases} x = \iota_1 \; x' \wedge y = \iota_1 \; y' & \Rightarrow x' \sqsubseteq_{\mathsf{T}_1} y' \\ x = \iota_2 \; x' \wedge y = \iota_2 \; y' & \Rightarrow x' \sqsubseteq_{\mathsf{T}_2} y' \end{cases} \\
x \sqsubseteq_{\mathsf{T}^K} y & \text{iff} & \forall_{k \in K} . \; x \; k \sqsubseteq_{\mathsf{T}} y \; k \\
x \sqsubseteq_{\mathcal{P}\mathsf{T}} y & \text{iff} & \forall_{e \in x} \exists_{e' \in y} . \; e \sqsubseteq_{\mathsf{T}} e'
\end{array}$$

where $\leq$ used above is the converse of relation $\geq$ induced in (3.4) by $\oplus$ in the underlying Q-algebra.

Based on this preorder, a *refinement* relation can be established between components with a cost as follows:

Definition 5.2. Given two components $q, p : I \longrightarrow O$, with costs in C, q refines p (*i.e.*, 'costs less than'), denoted by $q \trianglelefteq p$, iff there exists a function $h : U_q \longrightarrow U_p$ such that

$$\mathsf{B}(h \times \mathsf{id}) \cdot \overline{a}_q \; \dot{\sqsubseteq}_{(\mathsf{B}(\mathsf{Id} \times (O \times C)))^I} \; \overline{a}_p \cdot h \tag{5.3}$$

or, equivalently,

$$\mathsf{B}(h \times \mathsf{id}) \cdot a_q \; \dot{\sqsubseteq}_{\mathsf{B}(\mathsf{Id} \times (O \times C))} \; a_p \cdot (h \times \mathsf{id}) \tag{5.4}$$

where $\sqsubseteq_{\mathsf{B}(\mathsf{Id} \times (O \times C))}$ is the cost reduction preorder given in definition 5.1.

In coalgebraic refinement, as discussed in [22], a preorder underlying a refinement relation cannot be arbitrary. In particular, it has to be compatible with the *structural membership* $\in_{\mathsf{T}}$ defined, in [13], by induction on the structure of regular functor T as follows:

$$\begin{aligned}
\in_{\mathsf{Id}} &= id \\
\in_{\mathsf{K}} &= \perp \\
\in_{\mathsf{T}_1 \times \mathsf{T}_2} &= (\in_{\mathsf{T}_1} \cdot \pi_1) \cup (\in_{\mathsf{T}_2} \cdot \pi_2) \\
\in_{\mathsf{T}_1 + \mathsf{T}_2} &= [\in_{\mathsf{T}_1}, \in_{\mathsf{T}_2}] \\
\in_{\mathsf{T}_1 \cdot \mathsf{T}_2} &= \in_{\mathsf{T}_2} \cdot \in_{\mathsf{T}_1} \\
\in_{\mathsf{T}^K} &= \bigcup_{k \in K} \in_{\mathsf{T}} \cdot \beta_k \text{ (where } \beta_k f = f\ k) \\
\in_{\mathcal{P}} &= \in \text{ (set-theoretic membership)}
\end{aligned}$$

This means that

$$(\in_{\mathsf{T}} \cdot \sqsubseteq_{\mathsf{T}}) \subseteq \in_{\mathsf{T}} \tag{5.5}$$

Therefore,

Lemma 5.1. *The cost reduction preorder given in definition* 5.1 *is compatible with* $\in_{\mathsf{T}}$, i.e.*, satisfies* (5.5).

Proof. Condition (5.5) is equivalent to

$$\sqsubseteq_{\mathsf{T}} \subseteq (\in_{\mathsf{T}} \backslash \in_{\mathsf{T}}) \tag{5.6}$$

which establishes the relational division of structural membership by itself as the greatest refinement preorder (see [22]). The latter is known as *structural inclusion* and denoted by $\subseteq_{\mathsf{T}}$. Note that $\sqsubseteq_{\mathsf{T}}$ differs from $\subseteq_{\mathsf{T}}$ only in the case $\mathsf{T} = K$. Now,

$$\begin{aligned}
& \in_K \backslash \in_K \\
= & \quad \{ \in_K = \perp \} \\
& \perp \backslash \perp \\
= & \quad \{ \text{relational division} \} \\
& \top
\end{aligned}$$

and, clearly, $\sqsubseteq_K \subseteq \top$.

□

The following result establishes monotonicity of refinement thus paving the way for modular reasoning.

Lemma 5.2. *The refinement relation* $\trianglelefteq$ *given in definition* 5.2 *is preserved by all component combinators in the calculus.*

Proof. We consider the cases of *parallel* and *sequential* composition. Preservation of $\trianglelefteq$ by the remaining operators is proved similarly. Suppose $p' \trianglelefteq p$ and $q' \trianglelefteq q$, which means there exists morphisms $h : U_p \longrightarrow U_{p'}$ and $k : U_q \longrightarrow U_{q'}$ such that

$$\mathsf{B}(h \times \mathsf{id}) \cdot a_{p'} \ \dot{\sqsubseteq}_{\mathsf{B}(\mathsf{Id} \times (O \times C))} \ a_p \cdot (h \times \mathsf{id}) \tag{5.7}$$

$$\mathsf{B}(k \times \mathsf{id}) \cdot a_{q'} \ \dot{\sqsubseteq}_{\mathsf{B}(\mathsf{Id} \times (O \times C))} \ a_q \cdot (k \times \mathsf{id}) \tag{5.8}$$

To prove that $p' \boxtimes q' \trianglelefteq p \boxtimes q$, take $h \times k$ as the witness morphism (*i.e.*, show that $h \times k$ is a forward morphism). Thus,

$$\begin{array}{cl}
& \mathsf{B}((h \times k) \times \mathsf{id}) \cdot a_{p' \boxtimes q'} \\
= & \quad \{ \ \boxtimes \text{ definition } \} \\
& \mathsf{B}((h \times k) \times \mathsf{id}) \cdot \mathsf{B}(\mathsf{id} \times (\oplus \times \mathsf{id})) \cdot \mathsf{B}(\mathsf{id} \times \mathsf{m}) \cdot \mathsf{Bm} \cdot \delta_l \cdot (a_{p'} \times a_{q'}) \cdot \mathsf{m} \\
= & \quad \{ \ \mathsf{m} \text{ natural, functoriality} \} \\
& \mathsf{B}(\mathsf{id} \times (\oplus \times \mathsf{id})) \cdot \mathsf{B}(\mathsf{id} \times \mathsf{m}) \cdot \mathsf{Bm} \cdot \mathsf{B}((h \times \mathsf{id}) \times (k \times \mathsf{id})) \cdot \delta_l \cdot (a_{p'} \times a_{q'}) \cdot \mathsf{m} \\
= & \quad \{ \ \delta_l \text{ natural } \} \\
& \mathsf{B}(\mathsf{id} \times (\oplus \times \mathsf{id})) \cdot \mathsf{B}(\mathsf{id} \times \mathsf{m}) \cdot \mathsf{Bm} \cdot \delta_l \cdot (\mathsf{B}(h \times \mathsf{id}) \cdot a_{p'} \times \mathsf{B}(k \times \mathsf{id}) \cdot a_{q'}) \cdot \mathsf{m} \\
\dot{\sqsubseteq} & \quad \{ \text{assumptions (5.7) and (5.8)} \} \\
& \mathsf{B}(\mathsf{id} \times (\oplus \times \mathsf{id})) \cdot \mathsf{B}(\mathsf{id} \times \mathsf{m}) \cdot \mathsf{Bm} \cdot \delta_l \cdot (a_p \times a_q) \cdot ((h \times \mathsf{id}) \times (k \times \mathsf{id})) \cdot \mathsf{m} \\
= & \quad \{ \ \mathsf{m} \text{ natural } \} \\
& \mathsf{B}(\mathsf{id} \times (\oplus \times \mathsf{id})) \cdot \mathsf{B}(\mathsf{id} \times \mathsf{m}) \cdot \mathsf{Bm} \cdot \delta_l \cdot (a_p \times a_q) \cdot \mathsf{m} \cdot ((h \times k) \times \mathsf{id}) \\
= & \quad \{ \ \boxtimes \text{ definition } \} \\
& a_{p \boxtimes q} \cdot ((h \times k) \times \mathsf{id})
\end{array}$$

Consider now sequential composition. We show that $p' \,;\, q' \trianglelefteq p \,;\, q$ holds by verifying that $h \times k$, again in this case, is a forward morphism. Thus,

$$\begin{array}{cl}
& \mathsf{B}((h \times k) \times \mathsf{id})) \cdot a_{p';q'} \\
= & \quad \{ \ ; \text{ definition } \} \\
& \mathsf{B}((h \times k) \times \mathsf{id})) \cdot \mathsf{B}(\mathsf{id} \times (\otimes \times \mathsf{id})) \cdot \mathsf{B}(\mathsf{id} \times \mathsf{a}^\circ) \cdot \mathsf{Bm} \cdot \mu \cdot \mathsf{B}\tau_l \cdot \mathsf{B}(\mathsf{id} \times a_{q'}) \\
& \cdot \mathsf{B}(\mathsf{m} \cdot \mathsf{xr}) \cdot \tau_r \cdot (a_{p'} \times \mathsf{id}) \cdot \mathsf{xr} \\
= & \quad \{ \ \text{functoriality } \} \\
& \mathsf{B}(\mathsf{id} \times (\otimes \times \mathsf{id})) \cdot \mathsf{B}((h \times k) \times ((\mathsf{id} \times \mathsf{id}) \times \mathsf{id})) \cdot \mathsf{B}(\mathsf{id} \times \mathsf{a}^\circ) \cdot \mathsf{Bm} \cdot \mu \cdot \mathsf{B}\tau_l \cdot \mathsf{B}(\mathsf{id} \times a_{q'}) \\
& \cdot \mathsf{B}(\mathsf{m} \cdot \mathsf{xr}) \cdot \tau_r \cdot (a_{p'} \times \mathsf{id}) \cdot \mathsf{xr} \\
= & \quad \{ \ \text{functoriality, } \mathsf{a}^\circ \text{ natural } \} \\
& \mathsf{B}(\mathsf{id} \times (\otimes \times \mathsf{id})) \cdot \mathsf{B}(\mathsf{id} \times \mathsf{a}^\circ) \cdot \mathsf{B}((h \times k) \times (\mathsf{id} \times (\mathsf{id} \times \mathsf{id})) \cdot \mathsf{Bm} \cdot \mu \cdot \mathsf{B}\tau_l \cdot \mathsf{B}(\mathsf{id} \times a_{q'}) \\
& \cdot \mathsf{B}(\mathsf{m} \cdot \mathsf{xr}) \cdot \tau_r \cdot (a_{p'} \times \mathsf{id}) \cdot \mathsf{xr} \\
= & \quad \{ \ \mu, \mathsf{m} \text{ natural } \} \\
& \mathsf{B}(\mathsf{id} \times (\otimes \times \mathsf{id})) \cdot \mathsf{B}(\mathsf{id} \times \mathsf{a}^\circ) \cdot \mathsf{Bm} \cdot \mu \cdot \mathsf{BB}((h \times \mathsf{id}) \times (k \times \mathsf{id})) \cdot \mathsf{B}\tau_l \cdot \mathsf{B}(\mathsf{id} \times a_{q'}) \\
& \cdot \mathsf{B}(\mathsf{m} \cdot \mathsf{xr}) \cdot \tau_r \cdot (a_{p'} \times \mathsf{id}) \cdot \mathsf{xr} \\
= & \quad \{ \ \tau_l \text{ natural } \} \\
& \mathsf{B}(\mathsf{id} \times (\otimes \times \mathsf{id})) \cdot \mathsf{B}(\mathsf{id} \times \mathsf{a}^\circ) \cdot \mathsf{Bm} \cdot \mu \cdot \mathsf{B}\tau_l \cdot \mathsf{B}((h \times \mathsf{id}) \times (\mathsf{B}(k \times \mathsf{id})) \cdot \mathsf{B}(\mathsf{id} \times a_{q'}) \\
& \cdot \mathsf{B}(\mathsf{m} \cdot \mathsf{xr}) \cdot \tau_r \cdot (a_{p'} \times \mathsf{id}) \cdot \mathsf{xr} \\
= & \quad \{ \text{functoriality} \}
\end{array}$$

$$
\begin{array}{ll}
& \mathsf{B}(\mathsf{id} \times (\otimes \times \mathsf{id})) \cdot \mathsf{B}(\mathsf{id} \times \mathsf{a}^\circ) \cdot \mathsf{Bm} \cdot \mu \cdot \mathsf{B}\tau_l \cdot \mathsf{B}((h \times \mathsf{id}) \times (\mathsf{B}(k \times \mathsf{id}) \cdot a_{q'}) \cdot \mathsf{B}(\mathsf{m} \cdot \mathsf{xr}) \\
& \cdot \tau_r \cdot (a_{p'} \times \mathsf{id}) \cdot \mathsf{xr} \\
\dot{\sqsubseteq} & \quad \{ \text{ assumption (5.8)} \} \\
& \mathsf{B}(\mathsf{id} \times (\otimes \times \mathsf{id})) \cdot \mathsf{B}(\mathsf{id} \times \mathsf{a}^\circ) \cdot \mathsf{Bm} \cdot \mu \cdot \mathsf{B}\tau_l \cdot \mathsf{B}((h \times \mathsf{id}) \times (a_q \cdot (k \times \mathsf{id}))) \cdot \mathsf{B}(\mathsf{m} \cdot \mathsf{xr}) \\
& \cdot \tau_r \cdot (a_{p'} \times \mathsf{id}) \cdot \mathsf{xr} \\
= & \quad \{ \text{ functoriality} \} \\
& \mathsf{B}(\mathsf{id} \times (\otimes \times \mathsf{id})) \cdot \mathsf{B}(\mathsf{id} \times \mathsf{a}^\circ) \cdot \mathsf{Bm} \cdot \mu \cdot \mathsf{B}\tau_l \cdot \mathsf{B}(\mathsf{id} \times a_q) \cdot \mathsf{B}((h \times \mathsf{id}) \times (k \times \mathsf{id})) \\
& \cdot \mathsf{B}(\mathsf{m} \cdot \mathsf{xr}) \cdot \tau_r \cdot (a_{p'} \times \mathsf{id}) \cdot \mathsf{xr} \\
= & \quad \{ \ \tau_r, \mathsf{m}, \mathsf{xr} \text{ natural } \} \\
& \mathsf{B}(\mathsf{id} \times (\otimes \times \mathsf{id})) \cdot \mathsf{B}(\mathsf{id} \times \mathsf{a}^\circ) \cdot \mathsf{Bm} \cdot \mu \cdot \mathsf{B}\tau_l \cdot \mathsf{B}(\mathsf{id} \times a_q) \cdot \mathsf{B}(\mathsf{m} \cdot \mathsf{xr}) \cdot \tau_r \\
& \cdot (\mathsf{B}(h \times \mathsf{id}) \times k) \cdot (a_{p'} \times \mathsf{id}) \cdot \mathsf{xr} \\
= & \quad \{ \text{ functoriality} \} \\
& \mathsf{B}(\mathsf{id} \times (\otimes \times \mathsf{id})) \cdot \mathsf{B}(\mathsf{id} \times \mathsf{a}^\circ) \cdot \mathsf{Bm} \cdot \mu \cdot \mathsf{B}\tau_l \cdot \mathsf{B}(\mathsf{id} \times a_q) \cdot \mathsf{B}(\mathsf{m} \cdot \mathsf{xr}) \cdot \tau_r \\
& \cdot ((\mathsf{B}(h \times \mathsf{id}) \cdot a_{p'}) \times k) \cdot \mathsf{xr} \\
\dot{\sqsubseteq} & \quad \{ \text{ assumption (5.7)} \} \\
& \mathsf{B}(\mathsf{id} \times (\otimes \times \mathsf{id})) \cdot \mathsf{B}(\mathsf{id} \times \mathsf{a}^\circ) \cdot \mathsf{Bm} \cdot \mu \cdot \mathsf{B}\tau_l \cdot \mathsf{B}(\mathsf{id} \times a_q) \cdot \mathsf{B}(\mathsf{m} \cdot \mathsf{xr}) \cdot \tau_r \\
& \cdot ((a_p \cdot (h \times \mathsf{id})) \times k) \cdot \mathsf{xr} \\
= & \quad \{ \ \mathsf{xr} \text{ natural, functoriality} \} \\
& \mathsf{B}(\mathsf{id} \times (\otimes \times \mathsf{id})) \cdot \mathsf{B}(\mathsf{id} \times \mathsf{a}^\circ) \cdot \mathsf{Bm} \cdot \mu \cdot \mathsf{B}\tau_l \cdot \mathsf{B}(\mathsf{id} \times a_q) \cdot \mathsf{B}(\mathsf{m} \cdot \mathsf{xr}) \cdot \tau_r \cdot (a_p \times \mathsf{id}) \\
& \cdot \mathsf{xr} \cdot ((h \times k) \times \mathsf{id}) \\
= & \quad \{ \ ; \text{ definition } \} \\
& a_{p;q} \cdot ((h \times k) \times \mathsf{id})
\end{array}
$$

□

This refinement relation based on the *cost reduction* preorder can be used to establish a relationship between different implementations of the Folder component discussed in the previous section. In particular, given that PlainStack $\trianglelefteq$ CostStack, it is immediate to conclude, by lemma 5.2, that

$$(((\mathsf{LeftS} \boxplus \mathsf{RightS})[\mathsf{wi},\mathsf{wo}]) \urcorner_{P+P})[\mathsf{fi},\mathsf{fo}] \trianglelefteq (((\mathsf{CostLeftS} \boxplus \mathsf{RightS})[\mathsf{wi},\mathsf{wo}]) \urcorner_{P+P})[\mathsf{fi},\mathsf{fo}]$$

where CostLeftS is an implementation of a left stack from a CostStack, while LeftS stands for an implementation based on PlainLeftS

5.2. The calculus revisited

The remaining of this section illustrates the calculus of QoS-aware components discussing which laws of the plain calculus are preserved, invalidated or transformed into refinement inequalities.

Component's wrapping and their composition with the lifting of the wrapping functions are bisimilar patterns in the plain calculus. But what is the case in this new, QoS-aware setting? Clearly, property

$$(p[f,g])[f',g'] \sim p[f\cdot f', g'\cdot g] \tag{5.9}$$

still holds as in both cases the associated cost is that of p. On the other hand, property

$$p[f,g] \sim \ulcorner f \urcorner \,;\, p \,;\, \ulcorner g \urcorner \tag{5.10}$$

only holds if functions f and g are lifted at no cost, *i.e.*, with perfect QoS. This means the associated cost value is $\mathbf{1}$ — the identity for sequential and parallel cost aggregation — or, more generally, whenever $C = \mathbf{1}$ is assumed. In general, the interplay between wrapping and function lifting is expressed by a refinement law in this new setting. The following lemma states the basic result.

Lemma 5.3. *For component* p *and function* f *suitably typed,*

$$p[f,\mathsf{id}] \;\trianglelefteq\; \ulcorner f \urcorner \,;\, p \tag{5.11}$$

Proof. Recall that in the plain calculus both components were bisimilar, a fact established by proving that, in such a setting, canonical (Set-)isomorphism $\mathsf{r}^\circ : U \longrightarrow \mathbf{1}\times U$ was still a morphism from $p[f,\mathsf{id}]$ to $\ulcorner f \urcorner \,;\, p$. In the presence of *cost*-components, we reason as follows:

$$
\begin{array}{ll}
& a_{\ulcorner f \urcorner ; p} \cdot (\mathsf{r}^\circ \times \mathsf{id}) \\
= & \quad \{ \text{ ; and function lifting definitions } \} \\
& \mathsf{B}(\mathsf{id}\times(\otimes\times\mathsf{id}))\cdot\mathsf{B}(\mathsf{id}\times\mathsf{a}^\circ)\cdot\mathsf{Bm}\cdot\mu\cdot\mathsf{B}\tau_l\cdot\mathsf{B}(\mathsf{id}\times a_p) \\
& \cdot\mathsf{Bm}\cdot\mathsf{Bxr}\cdot\tau_r\cdot(\eta\times\mathsf{id})\cdot(\mathsf{id}\times\langle \underline{c}, f\rangle)\times\mathsf{id}\cdot\mathsf{xr}\cdot(\mathsf{r}^\circ\times\mathsf{id}) \\
= & \quad \{ \; \eta \text{ strong } (\eta = \tau_r\cdot(\eta\times\mathsf{id})), \text{ natural } (\mathsf{B}f\cdot\eta = \eta\cdot f) \} \\
& \mathsf{B}(\mathsf{id}\times(\otimes\times\mathsf{id}))\cdot\mathsf{B}(\mathsf{id}\times\mathsf{a}^\circ)\cdot\mathsf{Bm}\cdot\mu\cdot\eta\cdot\tau_l\cdot(\mathsf{id}\times a_p) \\
& \cdot\mathsf{m}\cdot\mathsf{xr}\cdot(\mathsf{id}\times\langle \underline{c}, f\rangle)\times\mathsf{id}\cdot\mathsf{xr}\cdot(\mathsf{r}^\circ\times\mathsf{id}) \\
= & \quad \{ \text{ monad laws: } \mu\cdot\eta = \mathsf{id} \} \\
& \mathsf{B}(\mathsf{id}\times(\otimes\times\mathsf{id}))\cdot\mathsf{B}(\mathsf{id}\times\mathsf{a}^\circ)\cdot\mathsf{Bm}\cdot\tau_l\cdot(\mathsf{id}\times a_p) \\
& \cdot\mathsf{m}\cdot\mathsf{xr}\cdot(\mathsf{id}\times\langle \underline{c}, f\rangle)\times\mathsf{id}\cdot\mathsf{xr}\cdot(\mathsf{r}^\circ\times\mathsf{id}) \\
= & \quad \{ \; \mathsf{xr} \text{ natural isomorphism and idempotent } \} \\
& \mathsf{B}(\mathsf{id}\times(\otimes\times\mathsf{id}))\cdot\mathsf{B}(\mathsf{id}\times\mathsf{a}^\circ)\cdot\mathsf{Bm}\cdot\tau_l\cdot(\mathsf{id}\times a_p)\cdot\mathsf{m}\cdot(\mathsf{id}\times\langle \underline{c}, f\rangle)\cdot(\mathsf{r}^\circ\times\mathsf{id}) \\
= & \quad \{ \text{ routine } (\mathsf{m}, \mathsf{xl}, \mathsf{r} \text{ natural isomorphisms, functoriality}) \} \\
& \mathsf{B}(\mathsf{id}\times(\otimes\times\mathsf{id}))\cdot\mathsf{B}(\mathsf{id}\times\mathsf{a}^\circ)\cdot\mathsf{Bm}\cdot\tau_l\cdot(\mathsf{id}\times a_p)\cdot(\mathsf{r}^\circ\times\mathsf{id})\cdot\mathsf{xl}\cdot(\mathsf{id}\times\langle \underline{c}, f\rangle) \\
= & \quad \{ \text{ functoriality } \} \\
& \mathsf{B}(\mathsf{id}\times(\otimes\times\mathsf{id}))\cdot\mathsf{B}(\mathsf{id}\times\mathsf{a}^\circ)\cdot\mathsf{Bm}\cdot\tau_l\cdot(\mathsf{r}^\circ\times\mathsf{id})\cdot(\mathsf{id}\times a_p)\cdot\mathsf{xl}\cdot(\mathsf{id}\times\langle \underline{c}, f\rangle) \\
= & \quad \{ \text{ routine } (\mathsf{xl} \text{ natural isomorphism, functoriality}) \} \\
& \mathsf{B}(\mathsf{id}\times(\otimes\times\mathsf{id}))\cdot\mathsf{B}(\mathsf{id}\times\mathsf{a}^\circ)\cdot\mathsf{Bm}\cdot\tau_l\cdot(\mathsf{r}^\circ\times\mathsf{id})\cdot(\mathsf{id}\times a_p)\cdot\langle \underline{c}, \mathsf{id}\rangle\cdot(\mathsf{id}\times f) \\
= & \quad \{ \text{ routine (split laws) } \}
\end{array}
$$

$$
\begin{array}{ll}
& \mathsf{B}(\mathsf{id} \times (\otimes \times \mathsf{id})) \cdot \mathsf{B}(\mathsf{id} \times \mathsf{a}^{\circ}) \cdot \mathsf{Bm} \cdot \tau_l \cdot (\mathsf{r}^{\circ} \times \mathsf{id}) \cdot \langle \underline{c}, \mathsf{id} \rangle \cdot a_p \cdot (\mathsf{id} \times f) \\
= & \quad \{ \; \tau_l \text{ natural } (\tau_l \cdot (h \times h') = \mathsf{B}(h \times h') \cdot \tau_l) \} \\
& \mathsf{B}(\mathsf{id} \times (\otimes \times \mathsf{id})) \cdot \mathsf{B}(\mathsf{id} \times \mathsf{a}^{\circ}) \cdot \mathsf{Bm} \cdot \mathsf{B}(\mathsf{r}^{\circ} \times \mathsf{id}) \cdot \tau_l \cdot \langle \underline{c}, \mathsf{id} \rangle \cdot a_p \cdot (\mathsf{id} \times f) \\
= & \quad \{ \; \tau_l \text{ naturality corollary } (\tau_l \cdot \langle \underline{c}, \mathsf{id} \rangle = \mathsf{B}\langle \underline{c}, \mathsf{id} \rangle) \} \\
& \mathsf{B}(\mathsf{id} \times (\otimes \times \mathsf{id})) \cdot \mathsf{B}(\mathsf{id} \times \mathsf{a}^{\circ}) \cdot \mathsf{Bm} \cdot \mathsf{B}(\mathsf{r}^{\circ} \times \mathsf{id}) \cdot \mathsf{B}\langle \underline{c}, \mathsf{id} \rangle \cdot a_p \cdot (\mathsf{id} \times f) \\
= & \quad \{ \; \text{routine } (\mathsf{r}, \mathsf{a} \text{ natural isomorphisms, functoriality}) \} \\
& \mathsf{B}(\mathsf{id} \times (\otimes \times \mathsf{id})) \cdot \mathsf{B}(\mathsf{id} \times \mathsf{a}^{\circ}) \cdot \mathsf{Bm} \cdot \mathsf{B}(\langle \underline{c}, \mathsf{id} \rangle \times \mathsf{id}) \cdot \mathsf{Ba} \cdot \mathsf{B}(\mathsf{r}^{\circ} \times \mathsf{id}) \cdot a_p \cdot (\mathsf{id} \times f) \\
= & \quad \{ \; \text{definition of } a_{p[f,\mathsf{id}]} \; \} \\
& \mathsf{B}(\mathsf{id} \times (\otimes \times \mathsf{id})) \cdot \mathsf{B}(\mathsf{id} \times \mathsf{a}^{\circ}) \cdot \mathsf{Bm} \cdot \mathsf{B}(\langle \underline{c}, \mathsf{id} \rangle \times \mathsf{id}) \cdot \mathsf{Ba} \cdot \mathsf{B}(\mathsf{r}^{\circ} \times \mathsf{id}) \cdot a_{p[f,\mathsf{id}]} \\
= & \quad \{ \; (*) \; \} \\
& \mathsf{B}(\mathsf{id} \times \phi_c) \cdot \mathsf{B}(\mathsf{r}^{\circ} \times \mathsf{id}) \cdot a_{p[f,\mathsf{id}]} \\
\dot{\sqsupseteq} & \quad \{ \text{ if } c \neq \mathbf{1} \; \} \\
& \mathsf{B}(\mathsf{r}^{\circ} \times \mathsf{id}) \cdot a_{p[f,\mathsf{id}]}
\end{array}
$$

where justification in step $(*)$ amounts to observe that, for each value $c \in C$, function

$$(\mathsf{id} \times (\otimes \times \mathsf{id})) \cdot (\mathsf{id} \times \mathsf{a}^{\circ}) \cdot \mathsf{m} \cdot (\langle \underline{c}, \mathsf{id} \rangle \times \mathsf{id}) \cdot \mathsf{a} : U_p \times (C \times O) \longrightarrow U_p \times (C \times O)$$

is equivalent to ϕ_c defined by $\phi_c((*, u), (d, o)) = ((*, u), (c \otimes d, o)$, for any $u \in U_p$, $o \in O$ and $d \in C$. Therefore we conclude that

$$\mathsf{B}(\mathsf{r}^{\circ} \times \mathsf{id}) \cdot a_{p[f,\mathsf{id}]} \;\; \dot{\sqsubseteq} \;\; a_{\ulcorner f \urcorner ; p} \cdot (\mathsf{r}^{\circ} \times \mathsf{id}) \tag{5.12}$$

which establishes r° as a forward morphism witnessing (5.11).

□

We will revisit now a number of laws representative of each main block of laws in the component calculus. The first, easy observation is that the monoidal structure associated to sequential composition — laws (2.5) and (2.6) — is preserved.

Lemma 5.4. *For components p, q and r suitably typed,*

$$\mathsf{copy}_I \,;\, p \;\sim\; p \;\sim\; p \,;\, \mathsf{copy}_O \tag{5.13}$$

$$(p \,;\, q) \,;\, r \;\sim\; p \,;\, (q \cdot r) \tag{5.14}$$

Proof. First define identity $\mathsf{copy}_K : K \longrightarrow K$ as

$$\mathsf{copy}_K \;=\; \langle * \in \mathbf{1}, \eta \cdot (\mathsf{id} \times \langle \underline{\mathbf{1}}, \mathsf{id} \rangle) \rangle$$

for $\mathbf{1} \in C$ the identity of $\otimes$ in the underlying Q-algebra. Then (5.13) and (5.14) follow from $\langle C, \otimes, \mathbf{1} \rangle$ being a monoid.

□

A similar result, and for the same reasons, holds for parallel composition, once identity for $\boxtimes$, component idle : $\mathbf{1} \longrightarrow \mathbf{1}$, is defined by the lifting of the identity over the singleton set $\mathbf{1}$ at no cost. In general, we have,

Lemma 5.5. *For components p, p', q, q', r suitably typed in each expression,*

$$(p \boxtimes q) \boxtimes r \sim (p \boxtimes (q \boxtimes r))[\mathsf{a}, \mathsf{a}^\circ] \tag{5.15}$$

$$\mathsf{idle} \boxtimes p \sim p[\mathsf{r}, \mathsf{r}^\circ] \tag{5.16}$$

$$\mathsf{nil} \boxtimes p \sim \mathsf{nil}[\mathsf{zl}, \mathsf{zl}^\circ] \tag{5.17}$$

$$p \boxtimes q \sim (q \boxtimes p)[\mathsf{s}, \mathsf{s}] \quad \textit{if } \mathsf{B} \textit{ is a commutative monad} \tag{5.18}$$

Proof. The reader is referred to [2] for a proof of each of these laws in the plain calculus. Note the role of Set isomorphisms for associativity, commutativity, product right identity and zero are need to keep track of components' interfaces. For these equalities to hold it is necessary (and sufficient for (5.17)) that the lifting of identities underlying the definitions of idle, copy and nil, has neutral cost $\mathbf{1}$. Then, equations (5.15) and (5.16) hold because $\langle C, \oplus, \mathbf{1}\rangle$ is a monoid. Finally, (5.18) additionally requires $\oplus$ to be commutative.

□

Other equalities governing parallel composition of components reduce to refinement laws in the presence of costs. Consider, for example, the question of determining whether $\boxtimes$ lifts to a Cartesian product in the category whose objects are sets and arrows are components. The first step to address this question entails the need for defining the *split* of two components as follows

$$\langle p, q\rangle = \ulcorner \triangle \urcorner ; (p \boxtimes q) \quad \text{where} \quad \triangle = \langle id, id\rangle$$

i.e., exactly as the split of two functions (formally, the *universal* function to the Cartesian product). This definition, however, does not guarantee, in general, the commutativity of

$$\begin{tikzcd} & I \arrow[dl, "p"'] \arrow[d, "\langle p,q\rangle"] \arrow[dr, "q"] & \\ O & O \boxtimes R \arrow[l, "\ulcorner \pi_1 \urcorner"] \arrow[r, "\ulcorner \pi_2 \urcorner"'] & R \end{tikzcd} \tag{5.19}$$

Qualifier *in general* above means *for any monad* B; obviously (5.19) holds for *deterministic* components (*i.e.*, with $\mathsf{B} = \mathsf{Id}$). One may prove, however, that, for a broad range of (commutative) monads a *cancellation* law

$$p \sim \langle p, q\rangle ; \ulcorner \pi_1 \urcorner \tag{5.20}$$

holds. In the extended QoS-aware calculus, however, equation (5.20) is only a refinement

$$p \trianglelefteq \langle p, q\rangle ; \ulcorner \pi_1 \urcorner \tag{5.21}$$

because the right-hand side entails the execution of both p and q, thus $\oplus$-composing the respective costs.

Not all laws in the plain calculus, however, are preserved or relaxed to refinement inequalities in this new setting. Some simply do not hold any more. This may be illustrated with the following negative result (which did hold in the plain case for B a commutative monad).

Lemma 5.6. *For* cost-*components, combinator* $\boxtimes$ *is no longer a lax functor,* i.e.*, the following law does not hold in general:*

$$(p \boxtimes p')\,;\,(q \boxtimes q') \;\sim\; (p\,;\,q) \boxtimes (p'\,;\,q') \tag{5.22}$$

Proof. If execution of the relevant components are annotated with specific QoS values, those are composed through $\otimes$ and $\oplus$ whenever components are aggregated through **;** and $\boxtimes$, respectively. Thus, in order to validate (5.22), the underlying Q-algebra must satisfy equality

$$(x \oplus x') \otimes (y \oplus y') \;=\; (x \otimes y) \oplus (x' \otimes y') \tag{5.23}$$

Clearly, (5.23) does not hold in general. Even worse: for the Q-algebra introduced in Example 1, depending on the concrete values chosen, equality (5.23) can be oriented with $\geq$ in either direction.

□

The case of parallel composition of lifted functions is also instructive. In the plain calculus equality

$$\ulcorner f \times g \urcorner \;\sim\; \ulcorner f \urcorner \boxtimes \ulcorner g \urcorner \tag{5.24}$$

holds. In the presence of non trivial costs associated to function lifting such is no more the case. Actually,

Lemma 5.7. *Law* (5.24) *holds when the cost of lifting both* f *and* g *is the identity for* $\otimes$*, or when both functions are lifted with a common, but arbitrary cost and* $\otimes$*, in the particular* Q*-algebra considered, is idempotent.*

Proof. As in the plain calculus case, the law is proved by trying to show that $\mathsf{r} : \mathbf{1} \times \mathbf{1} \longrightarrow \mathbf{1}$ is a morphism from $\ulcorner f \urcorner \boxtimes \ulcorner g \urcorner$ to $\ulcorner f \times g \urcorner$. Suppose that both functions are lifted with a common, but arbitrary cost c. Then,

$$
\begin{array}{ll}
& \mathsf{B}(\mathsf{r} \times \mathsf{id}) \cdot a_{\ulcorner f \urcorner \boxtimes \ulcorner g \urcorner} \\
= & \quad \{\ \boxtimes \text{ definition and } \textit{function lifting}\ \} \\
& \mathsf{B}(\mathsf{r} \times \mathsf{id}) \cdot \mathsf{B}(\mathsf{id} \times (\oplus \times \mathsf{id})) \cdot \mathsf{B}(\mathsf{id} \times \mathsf{m}) \cdot \mathsf{Bm} \cdot \delta_l \cdot (\eta \times \eta) \\
& \cdot((\mathsf{id} \times \langle \underline{c}, f \rangle) \times (\mathsf{id} \times \langle \underline{c}, g \rangle)) \cdot \mathsf{m} \\
= & \quad \{\ \delta_l \cdot (\eta \times \eta) = \eta \} \\
& \mathsf{B}(\mathsf{r} \times \mathsf{id}) \cdot \mathsf{B}(\mathsf{id} \times (\oplus \times \mathsf{id})) \cdot \mathsf{B}(\mathsf{id} \times \mathsf{m}) \cdot \mathsf{Bm} \cdot \eta \cdot ((\mathsf{id} \times \langle \underline{c}, f \rangle) \times (\mathsf{id} \times \langle \underline{c}, g \rangle)) \cdot \mathsf{m} \\
= & \quad \{\ \eta \text{ natural} \} \\
& \eta \cdot (\mathsf{r} \times \mathsf{id}) \cdot (\mathsf{id} \times (\oplus \times \mathsf{id})) \cdot (\mathsf{id} \times \mathsf{m}) \cdot \mathsf{m} \cdot ((\mathsf{id} \times \langle \underline{c}, f \rangle) \times (\mathsf{id} \times \langle \underline{c}, g \rangle)) \cdot \mathsf{m} \\
= & \quad \{\ \mathsf{m} \text{ natural} \} \\
& \eta \cdot (\mathsf{r} \times \mathsf{id}) \cdot (\mathsf{id} \times (\oplus \times \mathsf{id})) \cdot (\mathsf{id} \times \mathsf{m}) \cdot ((\mathsf{id} \times \langle \underline{c}, f \rangle) \times (\mathsf{id} \times \langle \underline{c}, g \rangle)) \cdot \mathsf{m} \cdot \mathsf{m} \\
= & \quad \{\ \mathsf{m} = \mathsf{m}^\circ, \mathsf{r} \text{ natural}\ \} \\
& \eta \cdot (\mathsf{id} \times (\oplus \times \mathsf{id})) \cdot (\mathsf{id} \times \mathsf{m}) \cdot ((\mathsf{id} \times \langle \underline{c}, f \rangle) \times (\mathsf{id} \times \langle \underline{c}, g \rangle)) \cdot (\mathsf{r} \times \mathsf{id}) \\
= & \quad \{\ \langle x, y \rangle = (x \times y) \cdot \triangle\ \} \\
& \eta \cdot (\mathsf{id} \times (\oplus \times \mathsf{id})) \cdot (\mathsf{id} \times \mathsf{m}) \cdot ((\mathsf{id} \times ((\underline{c} \times f) \cdot \triangle)) \times (\mathsf{id} \times ((\underline{c} \times g)) \cdot \triangle))) \cdot (\mathsf{r} \times \mathsf{id}) \\
= & \quad \{\ \mathsf{m} \text{ natural, functoriality}\ \}
\end{array}
$$

$$
\begin{aligned}
& \eta \cdot (\mathsf{id} \times (\oplus \times \mathsf{id})) \cdot (\mathsf{id} \times ((\underline{c} \times \underline{c}) \times (f \times g))) \cdot (\mathsf{id} \times \mathsf{m}) \cdot (\mathsf{id} \times (\triangle \times \triangle)) \cdot (\mathsf{r} \times \mathsf{id}) \\
= & \quad \{ \ \mathsf{m} \cdot (\triangle \times \triangle) = \triangle, \text{functoriality} \ \} \\
& \eta \cdot (\mathsf{id} \times (\oplus \cdot (\underline{c} \times \underline{c}) \times (f \times g))) \cdot (\mathsf{id} \times \triangle) \cdot (\mathsf{r} \times \mathsf{id}) \\
= & \quad \{ \ \langle x, y \rangle = (x \times y) \cdot \triangle \ \} \\
& \eta \cdot (\mathsf{id} \times \langle \oplus \cdot (\underline{c} \times \underline{c}), (f \times g) \rangle) \cdot (\mathsf{r} \times \mathsf{id}) \\
= & \quad \{ \ \text{assumption: } \otimes \text{ idempotent} \ \} \\
& \eta \cdot (\mathsf{id} \times \langle \underline{c}, (f \times g) \rangle) \cdot (\mathsf{r} \times \mathsf{id}) \\
= & \quad \{ \ \textit{function lifting} \ \} \\
& a_{\ulcorner f \times g \urcorner} \cdot (\mathsf{r} \times \mathsf{id})
\end{aligned}
$$

Clearly, the same result is obtained if $\mathbf{1}$ is the cost associated to function lifting.

□

Note, however, that out of the conditions of the Lemma above, there is no hint on how to orient (5.24) as a refinement law.

We will not pursue here this line of inquiry: the study of the remaining combinators follows similar lines. The case for choice, combinator $\boxplus$, is particularly simple. Actually,

Lemma 5.8. *All equational laws exclusively involving the choice combinator in the plain calculus remain valid if cost-components are considered.*

Proof. Recalling $\boxplus$ definition (Definition 3.8), observe that, a bit surprisingly, it does not resort to any combinator of QoS values. On second thoughts, this is actually what one may expect: the QoS level associated to the execution of $p \boxplus q$ is either the QoS level of p or of q, depending on which alternative is chosen. What the Q-algebra gives, however, is a way of majoring this value: if $c_p, c_q \in C$ are the QoS levels associated to the execution of p and q, respectively, the upper bound for the execution of $p \boxplus q$ is $c_p \oplus c_q$.

□

6. Tool support

This section describes a prototype for the component calculus with QoS information developed as a *proof-of-concept*. It allows the (interactive) definition of state-based components through the set of combinators available in the calculus in their QoS-aware version as described in sections 2 and 3.

The tool is composed of an HASKELL combinator library and a graphical user interface developed in SWING. The choice of HASKELL was motivated by its expressiveness and extensibility, which provides an ideal means to support domain specific languages. HASKELL 'monadic technology' provides all the ingredients for a direct implementation of the calculus' combinators, suitably parametric on a strong monad `b`. Each component is represented by a monadic function from pairs of state-input values to `b`-computations of state-output pairs. The encoding of each combinator in the calculus follows closely the corresponding mathematical construction, as illustrated below for sequential composition.

```
seqComp :: Strong b => ((u,i)-> b (u,c,k)) -> ((v,k)-> b (v,c,o))
                    -> ((u,v), i) -> b ((u,v),c,o)

seqComp p q = (fmap id >< seqQoS >< id) . (fmap m >< id) . mult .
              (fmap (fmap converse(assocl))). (fmap lstr).
              (fmap (id >< q)) . (fmap id >< xr) . rstr .
              (p >< id) . xr
```

Operation seqQoS is part of the Q-algebra structure introduced in section 3 and corresponds to the sequential composition of QoS values, denoted by $\otimes$ above. It is the user responsibility to provide the Q-algebra suitable for the intended QoS analysis by instantiating the following data structure:

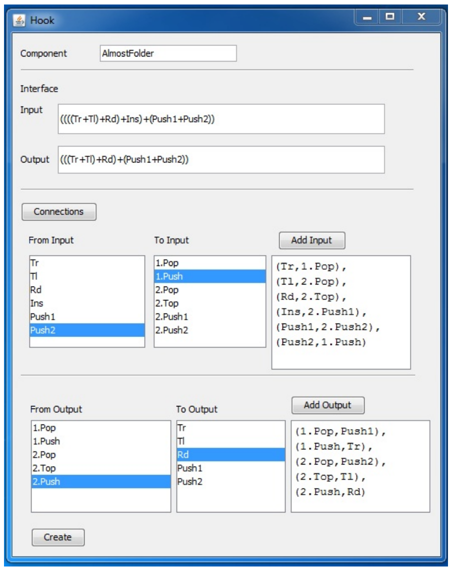
```
data Qalgebra v b a'=  QoS { vl :: v,
                             choiceQoS :: b-> b-> a',
                             seqQoS :: b-> b-> a',
                             concQoS :: b-> b-> a'
                            }  deriving (Typeable)
```

Computation proceeds through Kleisli composition. Note, finally, that in order to guarantee state persistence (and propagation of state values) the implementation resorts to the HASKELL state monad which is suitably combined with monad b capturing the underlying behavioral model. Integration with SWING, to provide a user-friendly interface, proved effective.

To give the reader a flavour of the tool, figure 1 illustrates the application of the *hook* combinator linking a number of ports with opposite polarity. It refers to the example discussed in Section 4 in which a *folder* component is built through the combination of two stacks modelling, respectively, the folder left and right piles.

Figure 2 depicts the result of this assembly process involving concurrent composition, wrapping and the establishment of a feedback loop, focussing on how internal and external components match. Wrapping components with suitable morphisms either for plug compatibility or to enforce specific port connections, as in this example, is a recurrent operation in the calculus, often error-prone when done manually. The prototyper allows this to be defined graphically, in an interactive way.

The prototyping tool allows both the (interactive) definition of component expressions and their execution in a simulation mode. Actually, once components are defined either from scratch (*i.e.*, by providing the corresponding code directly) or by composition of other components, the prototyper offers an environment for testing by simulation. The *Run* window in the tool offers two simulation modes: a *free* mode in which, if the component's behaviour model allows, may lead to 'disaster' (*e.g.*, by violation of port pre-conditions on a *partial* component), and a *safe* mode in which the effect of a port operation is forseen and eventually precluded.


Hook

Component: AlmostFolder

Interface

Input: ((((Tr+Tl)+Rd)+Ins)+(Push1+Push2))

Output: (((Tr+Tl)+Rd)+(Push1+Push2))

Connections

From Input	To Input
Tr	1.Pop
Tl	1.Push
Rd	2.Pop
Ins	2.Top
Push1	2.Push1
Push2	2.Push2

Add Input

```
(Tr,1.Pop),
(Tl,2.Pop),
(Rd,2.Top),
(Ins,2.Push1),
(Push1,2.Push2),
(Push2,1.Push)
```

From Output	To Output
1.Pop	Tr
1.Push	Tl
2.Pop	Rd
2.Top	Push1
2.Push	Push2

Add Output

```
(1.Pop,Push1),
(1.Push,Tr),
(2.Pop,Push2),
(2.Top,Tl),
(2.Push,Rd)
```

Create

FIGURE 1. Linking ports through the *hook* combinator

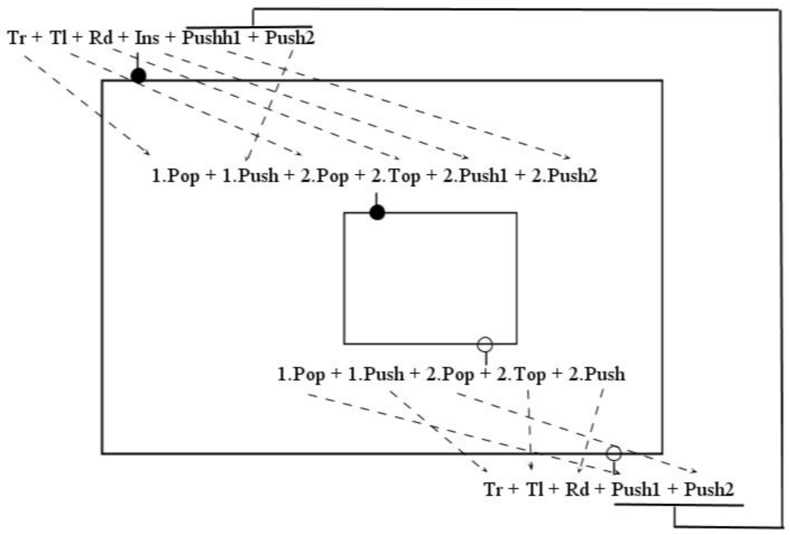
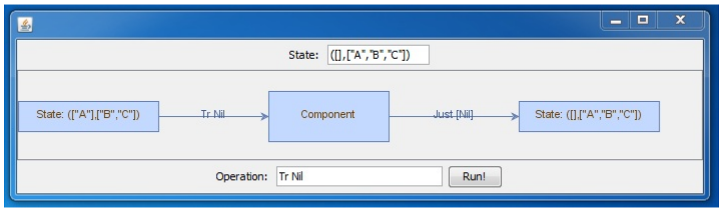

Tr + Tl + Rd + Ins + Pushh1 + Push2
1.Pop + 1.Push + 2.Pop + 2.Top + 2.Push1 + 2.Push2
1.Pop + 1.Push + 2.Pop + 2.Top + 2.Push
Tr + Tl + Rd + Push1 + Push2

FIGURE 2. Internal and external interface matching for the Folder component



State: ([],["A","B","C"])

Operation: Tr Nil

FIGURE 3. Component animation

Component testing, on the other hand, can be made in a purely interactive way, running event by event, or resorting to a *customising* expression. The calculus regards components as state-based entities interacting through well defined interfaces of observers and actions. It is often the case, however, that a particular component is *used* in a restricted way, namely as part of a broader system. This entails the need for a specification of the intended behaviour, which is not intrinsic to the component itself, but to its *role* (use) in a particular situation. For example, one may want to prescribe that action a is the initial action or that an action b is to follow each occurrence of a.

Such a process of *customising* a component's use is specified in the prototyping tool through an extended regular expression over invocations of the operations in the component interface. The extension is simply the inclusion of an interleaving operator, as in [25], which simplifies writing although it does not increase the original expressive power of regular expressions. Figure 3 illustrates the execution mode.

7. Conclusions

We have shown how, with a slight generalization of the notion of a Q-algebra proposed in [9], a component calculus can be extended, in a *systematic* way, to deal with QoS measures. This shows that reasoning formally about QoS-aware components is feasible; the resulting calculus being a smooth extension of the original one.

A new dimension of refinement seems to emerge from this work. It may be called *quality improving refinement* and is broadly defined as a way to guarantee not only a behavioural simulation, as in [22, 6], but also a higher (or at least equal) service quality.

The extended component calculus is doubled parametric in both dimensions: *behavior* (through a strong monad B) and *QoS* (through a Q-algebra). Therefore, it offers a suitable setting for reasoning, at a high level of abstraction, about component composition and, in general, coordination problems. Whether and how it scales up to composition of mobile components and their dynamic reconfiguration, is the topic of our current research.

The prototyping tool described in section 6 extends to the full calculus with QoS parameters, a preliminary tool was developed within the first author's research group and is documented in [16].

As future work, we are interested in exploiting the correspondence between the algebraic structure of a Q-algebra and the algebraic structure of the category of components where compositionality of behavior follows from the 'microcosm' structure in the sense of [10]. Further comparison of our results on quality-improving refinement with the generic notion of simulation proposed in [11, 12] and the approach based on Galois connections of [23] will be pursued.

References

[1] J. Adamek, *An introduction to coalgebra*. Theory and Applications of Categories **14**(8) (2005) 157–199.

[2] L. S. Barbosa, *Towards a Calculus of State-based Software Components*, J. of Universal Computer Science **9**(8) (2003), 891–909.

[3] L. S. Barbosa and Sun Meng, *QoS-aware Component Composition*, Proc. CISIS 2010, IEEE Computer Society (2010), 1008–1013.

[4] L. S. Barbosa, Sun Meng, B. K. Aichernig and N. Rodrigues. *On the semantics of componentware: A coalgebraic perspective*, In J. He and Z. Liu eds. Mathematical Frameworks for Component Software, World Scientific, 2006, 69–117.

[5] L. S. Barbosa and J. N. Oliveira, *State-based components made generic*, Electronic Notes in Theoretical Computer Science **82**(1) (2003), 39–56.

[6] L. S. Barbosa and J. N. Oliveira, *Transposing partial components: an exercise on coalgebraic refinement*, Theoretical Computer Science **365**(1-2) (2006) 2–22.

[7] R. Bird and O. de Moor, *Algebra of Programming*, Prentice Hall, 1997.

[8] G. Bolch, S. Greiner, H. de Meer and K. S. Trivedi, *Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications*, John Wiley, 2006.

[9] T. Chothia and J. Kleijn, *Q-Automata: Modelling the Resource Usage of Concurrent Components*, Electronic Notes in Theoretical Computer Science **175**(2) (2007), 153–167.

[10] I. Hasuo, C. Heunen, B. Jacobs and A. Sokolova, *Coalgebraic Components in a Many-Sorted Microcosm*. Proc. of CALCO 2009, Lecture Notes in Computer Science **5728**, Springer Verlag (2009), 64–80.

[11] J. Hughes and B. Jacobs, *Simulations in coalgebra*. Theoretical Computer Science **327**(1-2) (2004), 71–108.

[12] I. Hasuo, *Generic Forward and Backward Simulations II: Probabilistic Simulation*. Proc. of CONCUR 2010, Lecture Notes in Computer Science **6269**, Springer Verlag (2009), 447–461.

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

[14] B. Jacobs, *Exercises in coalgebraic specification*, Algebraic and Coalgebraic Methods in the Mathematics of Program Construction, volume 2297 of *LNCS*, Springer, 2002, 237–280.

[15] A. Kock, *Strong functors and monoidal monads*, Archiv für Mathematik **23** (1972), 113–120.

[16] A. Martins, L. S. Barbosa and N. F. Rodrigues, *Shacc: A Functional Prototyper for a Component Calculus*, Proc. CALCO-Tools'11, Lecture Notes in Computer Science **6859**, Springer Verlag (2011), 413–419.

[17] M. A. Marsan, G. Conte and G. Balbo, *A Class of Generalized Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems*, ACM Transactions on Computer Systems **2**(2) (1984), 93–122.

[18] D. A. Menascé, *Composing Web Services: A QoS View*, IEEE Internet Computing **8**(6) (2004), 88–90.

[19] R. Milner, *Communication and Concurrency*, Prentice Hall, 1989.

[20] J. Rutten, *Universal coalgebra: a theory of systems*, Theoretical Computer Science **249** (2000), 3–80.

[21] Sun Meng and F. Arbab, *QoS-Driven Service Selection and Composition Using Quantitative Constraint Automata*, Fundamenta Informaticae **95**(1) (2009) 103–128.

[22] Sun Meng and L. S. Barbosa, *Components as Coalgebras: the Refinement Dimension*, Theoretical Computer Science **351**(2) (2006), 276–294.

[23] Sun Meng, *Pre-Galois Connection on Coalgebras for Generic Component Refinement*, Electronic Notes in Theoretical Computer Science **207** (2008), 203-217.

[24] C. Szyperski, D. Gruntz and S. Murer, *Component Software - Beyond Object-Oriented Programming*, 2nd Edition, Publishing House of Electronics Industry, 2003.

[25] P. D. Stotts and W. Pugh, *Parallel finite automata for modeling concurrent software systems*. Journal of Systems and Software **27**(1) (1994) 27–43.

[26] K. Tarnay, *Protocol Specification and Testing*, Plenum Press, 1991.

[27] L. Zeng, B. Benatallah, A. H.H. Ngu, M. Dumas, J. Kalagnanam and H. Chang, *QoS-Aware Middleware for Web Services Composition*, IEEE Transactions on Software Engineering **30**(5) (2004), 311-327.

Acknowledgments

L. S. Barbosa was partially funded by ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT (Portuguese Foundation for Science and Technology) within project `FCOMP-01-0124-FEDER-0100478`. Sun Meng's work was partially supported by the National Natural Science Foundation of China under grant no. 61202069, and the Macao Science and Technology Development Fund under the PEARL project, grant number 041/2007/A3.

L. S. Barbosa
HASLab - INESC TEC and Department of Informatics, Universidade do Minho, Braga, Portugal
e-mail: `lsb@di.uminho.pt`

Sun Meng
Corresponding author.
LMAM, School of Mathematical Science, Peking University, Beijing, China, 100871
State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
e-mail: `sunmeng@math.pku.edu.cn`