Generic Components

(Extended Abstract)

Luis Barbosa and Sun Meng

U. Minho (Portugal) and Unu/Iist (Macau)

(lsb@di.uminho.pt, sm@iist.unu.edu)

Abstract. This extended abstract reports on the development of coalgebraic models for (state-based) software components, parametrized by a notion of *behaviour*. The latter is introduced as a strong (usually commutative) monad. Component operators and some of their laws are also discussed as a basis to reason about (and transform) state-based software designs.

1 Introduction

The expression *software component*, like many others in programming engineering, is so semantically overloaded that referring to it is often a risk. In this paper a *software component* is understood as the *specification* of a *state-based* module, eventually acting as a 'building block' of larger, often concurrent, systems. Again that qualification *concurrent* may convey several different meanings, *e.g.*, independent evolution, cooperation to achieve a common goal, competition for a shared resource. Such a component should encapsulate a number of services through a public *interface* which provides limited access to its internal state space. Furthermore, it *persists* and *evolves* in time.

The notion arises from a widespread paradigm for formally approaching systems' design: the so-called *model oriented* specification method, of which Vdm [11], Z [20] and Raise [24, 25] are well-known representatives. In such methods, an explicit *model* of the system being specified, rather than an axiomatic *theory* capturing its requirements, is defined. Usually a distinguished sort plays the role of a state space. State is often taken as a 'black box' accessible only via specified operations, which justifies the qualification of *state-based* typically given to such methods. Data and functionality are explicitly defined, while the temporal ordering of operation calls and, in general, most behavioural issues, are left implicit. Often the only behavioural information that can be extracted from, say, a Vdm specification amounts to the local constraints recorded in the operations' pre-conditions. This is, however, implicit and does not carry enough information neither on the systems architecture nor on the intended usage.

The emerging *component-oriented* programming paradigm [23, 26] retains from object-orientation the basic principle of encapsulation of data and code, but shifts the emphasis from (class) inheritance to (object) composition to avoid

interference between the former and encapsulation and, thus, paving the way to a development methodology based on *third-party assembly* of components.

However, as happened before with object-orientation (and even with programming in the broadest sense), component-orientation has grown up to a popular technology before consensual definitions and principles, let alone formal foundations, have been put forward [1].

This extended abstract provides an overview of the authors' on-going research on coalgebraic models for software components. Section 2 introduces the basic model which leads to the definition of a number of component combinators and the discussion of their laws in sections 3 and 4. An alternative, more expressive, model allowing for the combination of different behavioural aspects through monad composition, is glimpsed in section 5. Finally, section 6 points out some connections to related research and future work.

2 Generic Components

Our starting point is the conjunction of two key ideas: first, the 'black-box' characterisation of software components favours an *observational* semantics; secondly, the proposed constructions should be *generic* in the sense that they should not depend on a particular notion of component behaviour. Let us explain briefly both topics.

State-based modelling favours *observational* semantics: any two internal configurations should be identified wherever indistinguishable by observation. This is nicely captured by coalgebra theory [19]. A coalgebra for an (endo)functor T is a map $p : U \longrightarrow \mathsf{T}\ U$ which may be thought of as a transition structure, of *shape* T, on a set (the *carrier* or *state space*) U. The shape of T describes not only the way the state is (partially) accessed through the *observers*, but also how it evolves through *actions*. The lack of constructors forces equality to be replaced by *bisimilarity* (*i.e.*, equality with respect to the observation structure provided by T) and induction by *coinduction* as a proof principle.

The other key idea is the application of the so-called *functorial* approach to datatypes, originated in the work of the ADJ group in the early seventies [9, 8], to the area of state-based systems modelling. This approach provides a basis for *generic programming* [1] — a discipline based on type *polymorphism* [17], and *polytypism* [10], which raises the level of abstraction of the programming discourse in a way such that seemingly disparate techniques and algorithms are unified into idealised, kernel programming schemata.

Let I and O be sets acting as component interfaces. A component p with input I and output O, represented as $p : I \longrightarrow O$, is modelled as a *concrete*, seeded coalgebra for the following Set endofunctor:

$$\mathsf{T}^{\mathsf{B}} \ = \ \mathsf{B}(\mathsf{Id} \times O)^{I} \tag{1}$$

[1] As put by P. Wadler in a 1999 Seminar suggestively entitled 'Component-based Programming under different paradigms', *Just as Eskimos need fifty words for ice, perhaps we need many words for components.*

where B is a strong monad abstracting from each specific behaviour model. I.e., a pair $\langle u_p \in U_p, \overline{a}_p : U_p \longrightarrow \mathsf{B}(U_p \times O)^I \rangle$, where a specific value u_p of the state space U_p is taken as the coalgebra 'initial' or 'seed' value and the dynamics of the latter is captured by currying a state-transition function $a_p : U_p \times I \longrightarrow \mathsf{B}\ (U_p \times O)$. In this way, the computation of an action 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.

The use of strong monads as behaviour models for components is the source of *genericity* in the approach (see [4] for a detailed discussion) [2]. Some useful possibilities are:

- *Identity*, $\mathsf{B} = \mathsf{Id}$, yielding *total* and *deterministic* components.
- *Partiality*, *i.e.*, the possibility of deadlock or failure, captured by the maybe monad, $\mathsf{B} = \mathsf{Id} + \mathbf{1}$, as in the "stack" example above.
- *Non determinism*, introduced by the (finite) powerset monad, $\mathsf{B} = \mathcal{P}$.
- *Ordered non determinism*, based on the (finite) sequence monad, $\mathsf{B} = \mathsf{Id}^*$.
- Monoidal "labelling", with $\mathsf{B} = \mathsf{Id} \times M$. Note that, for B to form a monad, parameter M should support a monoidal structure to be used in the definition of η and μ.
- *'Metric' non determinism*, supported on a notion of a *bag* monad based on a structure $\langle M, \oplus, \otimes \rangle$, where both $\oplus$ and $\otimes$ define Abelian monoids over M and the latter distributes over the former. This captures situations in which, among the possible future evolutions of the component, some are more likely (or cheaper, more secure, *etc*) than others. See [4] for particular instantiations.

All of the above situations correspond to known strong monads in Set, which can be composed with each other. The first two and the last one are commutative; the third is not. Commutativity of 'monoidal labelling' depends, of course, on commutativity of the underlying monoid.

3 Components As Coalgebras

Let I and O be sets acting as component interfaces. A component p with input I and output O, represented as $p : I \longrightarrow O$, is modelled as a *concrete*, seeded

[2] 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 strong natural transformations [14]. 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. Strength τ_r, followed by τ_l maps $\mathsf{B}I \times \mathsf{B}J$ to $\mathsf{BB}(I \times J)$, which can, then, be flattened to $\mathsf{B}(I \times J)$ via μ. In most cases, however, the *order* of application is relevant for the outcome. The Kleisli composition[3] of the right with the left strength, gives rise to a natural transformation whose component on objects I and J is given by $\delta r_{I,J} = \tau_{r_{I,J}} \bullet \tau_{l_{\mathsf{B}I,J}}$ Dually, $\delta l_{I,J} = \tau_{l_{I,J}} \bullet \tau_{r_{I,\mathsf{B}J}}$. Such transformations specify how the monad distributes over product and, therefore, represent a sort of sequential composition of B-computations. Whenever δ_r and δ_l coincide, the monad is said to be *commutative*.

coalgebra for the following Set endofunctor:

$$\mathsf{T}^{\mathsf{B}} \;=\; \mathsf{B}(\mathsf{Id} \times O)^{I} \tag{2}$$

where B is a strong monad abstracting from each specific behaviour model[4]. I.e., a pair $\langle u_p \in U_p, \overline{a}_p : U_p \longrightarrow \mathsf{B}(U_p \times O)^{I}\rangle$, where a specific value u_p of the state space U_p is taken as the coalgebra 'initial' or 'seed' value and the dynamics of the latter is captured by currying a state-transition function $a_p : U_p \times I \longrightarrow \mathsf{B}\,(U_p \times O)$. In this way, the computation of an action 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.

Components "are arrows" and so arrows between components are "arrows between arrows", which motivates the adoption of a *bicategorical* [5] framework to structure our reasoning universe. In particular, a *2-cell* $h : p \longrightarrow q$ is a function relating the state spaces of p and q and satisfying the following *seed preservation* and *coalgebra* conditions:

$$h\, u_p \;=\; u_q \tag{3}$$

$$\overline{a}_q \cdot h \;=\; \mathsf{T}^{\mathsf{B}}\, h \cdot \overline{a}_p \tag{4}$$

2-cell 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 the carrier of p.
Let us denote by Cp such a bicategory. For each triple of objects $\langle I, K, O\rangle$, a composition law is given by a functor

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

The action of this on two objects p and q is given by

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

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[5]

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

[4] Such a parametrization is a source of *genericity* in the approach; see appendix A for a quick overview and [4] for a detailed discussion.

[5] The development, in a *point-free* style, of the component calculus discussed in this paper, resorts to a number of laws relating common 'housekeeping' morphisms to cope with *e.g.* product and sum associativity (a and a_+), commutativity (s), right and left units (r and l), right and left distributivity (dr, dl) or exchange (*i.e.*, morphisms $\mathsf{xl} : A \times (B \times C) \longrightarrow B \times (A \times C)$, $\mathsf{xr} : A \times B \times C \longrightarrow A \times C \times B$ and $\mathsf{m} : (A \times B) \times (C \times D) \longrightarrow (A \times C) \times (B \times D)$) with monad unit, multiplication and strength. Such laws are thoroughly dealt with in [2] under the designation of *context lemmas*.

The action of ";" on 2-cells is simply given by $h\,;k \;=\; h \times k$. Also notice that the definition above relies solely on properties of the monad morphisms η, μ, strengths τ_r and τ_l, and the distributive law δ_l. Finally, for each object K, an identity law is given by a functor

$$\mathsf{copy}_K : \mathbf{1} \longrightarrow \mathsf{Cp}(K, K)$$

whose action on objects is the constant component $\langle * \in \mathbf{1}, \overline{a}_{\mathsf{copy}_K} \rangle$, where $a_{\mathsf{copy}_K} = \eta_{\mathbf{1} \times K}$. Similarly, the action on morphisms is the constant comorphism $\mathsf{id}_{\mathbf{1}}$.

The fact that, for each strong monad B, components form a bicategory [6] amounts not only to a standard definition of the two basic combinators (";" and copy_K) of the component calculus, but also to setting up its basic laws:

$$\mathsf{copy}_I\,;p \;\sim\; p \;\sim\; p\,;\mathsf{copy}_O \tag{5}$$

$$(p\,;q)\,;r \;\sim\; p\,;(q\,;r) \tag{6}$$

Such laws are stated as T^{B}-bisimilarity equations, *i.e.*, up to observation through the 'shape' encoded in functor T^{B}.

Any function $f : A \longrightarrow B$ can be regarded as a (particular case of a) component, being lifted to Cp as

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

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

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

Clearly, function lifting is functorial. Moreover, isomorphisms, split monos and split epis lift to Cp as, respectively, isomorphisms, split monos and split epis [2].

The coalgebraic specification of a component specification describes immediate reactions to possible state/input configurations. It is its extension in time which gives the component's *behaviour* (abstracted as an element of the *final* coalgebra).. Formally, the behaviour $[\![p]\!]$ of a component p is computed by *coinductive extension* [19], *i.e.*, in the terminology of the 'Bird-Meertens calculi' style [6], by applying the induced *anamorphism* to the seed-value of p, *i.e.*, $[\![p]\!] = [\![\overline{a}_p]\!]u_p$.

Behaviours organise themselves in a category Bh whose objects are sets and each arrow $b : I \longrightarrow O$ is an element of $\nu_{I,O}$, the carrier of the final coalgebra $\omega_{I,O}$ for functor $\mathsf{B}(\mathsf{Id} \times O)^I$. Note 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 T^{B}-bisimulation. However, as final coalgebras are fully abstract with respect to bisimulation, the bicategorical structure collapses: the hom-categories become simply hom-*sets*. Of course, properties holding in Cp up to bisimulation, do hold 'on the nose' in the behaviour category.

[6] The reader is referred to [2] for all omitted proofs.

4 An Overview of the Component Calculus

Components can be aggregated in a number of different ways, besides the 'pipeline' composition discussed above. A calculus emerges from the the structure of Cp_{B} by introducing a 'wrapping' combinator, which may be thought of as an extension of the *renaming* connective found in process calculi (*e.g.*, [16]), and three tensors. Finally, generalized interaction is introduced through a sort of 'feedback' mechanism. Notice that the definition of all combinators is *parametric* on the *behaviour model*, relying on generic properties of the strong monad B. Component combinators are shown to be either lax endofunctors in Cp_{B} or simply functors between families of hom-categories. Their definitions carry naturally to Bh_{B}, where they show up as behaviour connectives, defining a particular (typed) 'process' algebra. In the sequel, and for illustrative purposes, we concentrate on three combinators: *wrapping*, *pipelining* and *choice* (see [4, 2] for a detailed exposition).

Wrapping

The pre- and post-composition of a component with Cp-lifted functions can be encapsulated in an unique *wrapping* combinator. Let $p : I \longrightarrow O$ be a component and consider functions $f : I' \longrightarrow I$ and $g : O \longrightarrow O'$. By $p[f, g]$ we will denote "component p wrapped by f and g". This has type $I' \longrightarrow O'$ and is defined by input pre-composition with f and output post-composition with g. Formally, the wrapping combinator is a functor (between the corresponding hom-categories)

$$-[f, g] : \mathsf{Cp}(I, O) \longrightarrow \mathsf{Cp}(I', O')$$

which is the identity on morphisms and maps a component $\langle u_p, \overline{a}_p \rangle$ into $\langle u_p, \overline{a}_{p[f,g]} \rangle$, where

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

Equations

$$p[f, g] \sim \ulcorner f \urcorner ; p ; \ulcorner g \urcorner \tag{7}$$

$$(p[f, g])[f', g'] \sim p[f \cdot f', g' \cdot g] \tag{8}$$

capture the basic properties about wrapping.

Some simple components arise by lifting elementary functions to Cp. We have already remarked that the lifting of the canonical arrow associated to the initial Set object plays the role of an *inert* component, unable to react to the outside world. Let us give this component a name:

$$\mathsf{inert}_A = \ulcorner ?_A \urcorner \tag{9}$$

In particular, we define the nil component

$$\mathsf{nil} = \mathsf{inert}_\emptyset = \ulcorner ?_\emptyset \urcorner = \ulcorner \mathsf{id}_\emptyset \urcorner \tag{10}$$

typed as $\mathsf{nil} : \emptyset \longrightarrow \emptyset$. Note that any component $p : I \longrightarrow O$ can be made inert by wrapping. For example, $p[?_I, !_O] \sim \mathsf{inert}_{\mathbf{1}}$.

A somewhat dual role is played by component $\mathsf{idle} = \ulcorner \mathsf{id}_{\mathbf{1}} \urcorner$. Note that $\mathsf{idle} : \mathbf{1} \longrightarrow \mathbf{1}$ is always willing to propagate an unstructured stimulus (*e.g.*, the push of a button) leading to a (similarly) unstructured reaction (*e.g.*, exciting a led).

Tensors

Three different composition patterns — *external choice* ($p \boxplus q$), *parallel* ($p \boxtimes q$) and *concurrent* composition ($p \circledast q$) — are captured by tensor products.

Let $p : I \longrightarrow O$ and $q : J \longrightarrow R$ be two components defined by $\langle u_p, \overline{a}_p \rangle$ and $\langle u_q, \overline{a}_q \rangle$, respectively. When interacting with $p \boxplus q : I + J \longrightarrow O + R$, the environment will be allowed to choose either to input a value of type I or one of type J, which will trigger the corresponding component (p or q, respectively), producing the relevant output. On its turn, *parallel* composition yields $p \boxtimes q : I \times J \longrightarrow O \times R$, corresponding to to a synchronous product: both components are executed simultaneously when triggered by a pair of legal input values. Note, however, that the behaviour effect, captured by monad B, propagates. For example, if B can express component failure and one of the arguments fails, the product will fail as well. Finally, *concurrent* composition combines choice and parallel, in the sense that p and q can be executed independently or jointly, depending on the input supplied.

The three combinators are defined as lax functors from $\mathsf{Cp} \times \mathsf{Cp}$ to Cp. Choice, for example, 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

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

where

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

which maps pairs of arrows $\langle h_1, h_2 \rangle$ into $h_1 \times h_2$. Parallel composition is similarly defined: just take $I \boxtimes J$ as $I \times J$ and

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

Finally, regarding $\boxast$ also as a lax functor, define $I \boxast J = I + J + I \times J$, and

$$
\begin{aligned}
a_{p \boxast q} = \quad & U_p \times U_q \times (I \boxast J) \xrightarrow{\mathsf{dr}} U_p \times U_q \times (I + J) + U_p \times U_q \times (I \times J) \\
& \xrightarrow{a_{p \boxplus q} + a_{p \boxtimes q}} \mathsf{B}\,(U_p \times U_q \times (O + R)) + \mathsf{B}\,(U_p \times U_q \times (O \times R)) \\
& \xrightarrow{[\mathsf{B}(\mathsf{id} \times \iota_1), \mathsf{B}(\mathsf{id} \times \iota_2)]} \mathsf{B}\,(U_p \times U_q \times (O \boxast R))
\end{aligned}
$$

Properties of these combinators are established in detail in [2]. Besides the verification of their definition as lax functors, it is shown that all of them are symmetric tensor products (commutativity of $\boxtimes$ and $\boxast$ depending on monad B being Abelian). In particular, nil is the unit of both $\boxplus$ and $\boxast$ and a zero element for $\boxtimes$, whereas idle is the unit for parallel composition. Thus,

$$
\begin{aligned}
(p \boxplus p') \,;\, (q \boxplus q') &\sim (p \,;\, q) \boxplus (p' \,;\, q') && (11) \\
\mathsf{copy}_{K \boxplus K'} &\sim \mathsf{copy}_K \boxplus \mathsf{copy}_{K'} && (12)
\end{aligned}
$$

$$
\begin{aligned}
(p \boxplus q) \boxplus r &\sim (p \boxplus (q \boxplus r))[\mathsf{a}_+, \mathsf{a}_+{}^\circ] && (13) \\
\mathsf{nil} \boxplus p &\sim p[\mathsf{r}_+, \mathsf{r}_+{}^\circ] && (14) \\
p \boxplus \mathsf{nil} &\sim p[\mathsf{l}_+, \mathsf{l}_+{}^\circ] && (15) \\
p \boxplus q &\sim (q \boxplus p)[\mathsf{s}_+, \mathsf{s}_+] && (16) \\
\mathsf{copy}_{K \boxtimes K'} &\sim \mathsf{copy}_K \boxtimes \mathsf{copy}_{K'} && (17) \\
(p \boxtimes p') \,;\, (q \boxtimes q') &\sim (p \,;\, q) \boxtimes (p' \,;\, q') \quad \text{(if } \mathsf{B} \text{ commutative)} && (18)
\end{aligned}
$$

$$
\begin{aligned}
(p \boxtimes q) \boxtimes r &\sim (p \boxtimes (q \boxtimes r))[\mathsf{a}, \mathsf{a}^\circ] && (19) \\
\mathsf{idle} \boxtimes p &\sim p[\mathsf{r}, \mathsf{r}^\circ] && (20) \\
\mathsf{nil} \boxtimes p &\sim \mathsf{nil}[\mathsf{zl}, \mathsf{zl}^\circ] && (21) \\
p \boxtimes q &\sim (q \boxtimes p)[\mathsf{s}, \mathsf{s}] \quad \text{(if } \mathsf{B} \text{ commutative)} && (22)
\end{aligned}
$$

Laws (23) and (24) relate $\boxplus$ and $\boxtimes$ with the usual Set sum and product, respectively, whereas laws (25) and (26) show them as specialisations of $\boxast$.

$$\ulcorner f \urcorner \boxplus \ulcorner g \urcorner \;\sim\; \ulcorner f + g \urcorner \tag{23}$$
$$\ulcorner f \urcorner \boxtimes \ulcorner g \urcorner \;\sim\; \ulcorner f \times g \urcorner \tag{24}$$
$$\ulcorner \iota_1 \urcorner ; (p \circledast q) \;\sim\; (p \boxplus q) ; \ulcorner \iota_1 \urcorner \tag{25}$$
$$\ulcorner \iota_2 \urcorner ; (p \circledast q) \;\sim\; (p \boxtimes q) ; \ulcorner \iota_2 \urcorner \tag{26}$$

Seeking for Universals

An important question for the development of a components' calculus is whether typical *universal* constructions used in (data-oriented) program calculi (see *e.g.*, [6]) have a counterpart here. Clearly, for any set I, the lifting of $?_I : \emptyset \longrightarrow I$ to Cp keeps naturality, *i.e.*, $\ulcorner ?_I \urcorner ; p \sim \ulcorner ?_O \urcorner$. As any bisimulation equation lifts to an equality in the behaviours category, $\emptyset$ is *initial* in Bh. For no trivial B, however, functions to $\mathbf{1}$ loose their naturality once lifted and, therefore, Bh has no final object.

Consider now combinators $\boxplus$ and $\boxtimes$. Is it possible to define counterparts in Cp to the *either* and *split* constructions in Set, by making[7]

$$[p, q] = (p \boxplus q) ; \ulcorner \nabla \urcorner \quad \text{and} \quad \langle p, q \rangle = \ulcorner \Delta \urcorner ; (p \boxtimes q) \ ?$$

The answer is only partially positive. In fact, codiagonal ∇ does not keep naturality once lifted to Cp, even for $\mathsf{B} = \mathsf{Id}$. Hence, uniqueness of component $[p, q]$ (see diagram below) is lost. However, *cancellation*, *reflection* and *absorption* laws still hold in Cp (equations (27), (28) and (29) respectively) and, therefore, $\boxplus$ becomes a weak coproduct in Bh. Also note that Set coproduct embeddings — once lifted to Cp, — keep their naturality (equations (31) and (32)), from where an 'idempotency' result (equation (33)) can be derived.

$$\begin{tikzcd} I \arrow[r, "\ulcorner \iota_1 \urcorner"] \arrow[dr, "p"'] & I \boxplus J \arrow[d, "{[p,q]}"] & J \arrow[l, "\ulcorner \iota_2 \urcorner"'] \arrow[dl, "q"] \\ & O & \end{tikzcd} \qquad \begin{array}{l} \ulcorner \iota_1 \urcorner ; [p, q] \sim p \\ \ulcorner \iota_2 \urcorner ; [p, q] \sim q \end{array} \tag{27}$$

$$[\ulcorner \iota_1 \urcorner, \ulcorner \iota_2 \urcorner] \;\sim\; \mathsf{copy}_{I+J} \tag{28}$$
$$(p \boxplus q) ; [p', q'] \;\sim\; [p ; p', q ; q'] \tag{29}$$

$$p \boxplus q \;\sim\; [p ; \ulcorner \iota_1 \urcorner, p ; \ulcorner \iota_2 \urcorner] \tag{30}$$
$$\ulcorner \iota_1 \urcorner ; (p \boxplus q) \;\sim\; p ; \ulcorner \iota_1 \urcorner \tag{31}$$
$$\ulcorner \iota_2 \urcorner ; (p \boxplus q) \;\sim\; q ; \ulcorner \iota_2 \urcorner \tag{32}$$
$$p ; \ulcorner \iota_1 \urcorner \;\sim\; \ulcorner \iota_1 \urcorner ; (p \boxplus p) \tag{33}$$

[7] where $\Delta = \langle \mathsf{id}, id \rangle : I \longrightarrow I \times I$ and $\nabla = [\mathsf{id}, id] : I + I \longrightarrow I$ are, respectively, the diagonal and codiagonal functions.

The dual situation, involving $\boxtimes$ and *split*, is a bit different. The problem here is that a *cancellation* result — $\langle p, q\rangle \,;\, \ulcorner\pi_1\urcorner \sim p$ — is only valid for a monad B which excludes the possibility of *failure* (*e.g.*, the non-empty powerset). On the other hand, diagonal Δ keeps its naturality when lifted to Cp, for B expressing deterministic behaviour (*e.g.*, the identity or the *maybe* monad), entailing a *fusion* law: $r \,;\, \langle p, q\rangle \sim \langle r \,;\, p, r \,;\, q\rangle$ Combining these two results, one concludes that $\boxtimes$ is a *product* in Bh, but only for behaviour models excluding failure and no determinism, which narrows the applicability scope of this fact to the category of total deterministic components. However, *reflection* and *absorption* laws (equations (34) and (35)) hold for any B.

$$\langle\ulcorner\pi_1\urcorner, \ulcorner\pi_2\urcorner\rangle \;\sim\; \mathsf{copy}_{O\times R} \tag{34}$$

$$\langle p, q\rangle \,;\, (p' \boxtimes q') \;\sim\; \langle p \,;\, p', q \,;\, q'\rangle \quad (\text{if } \mathsf{B} \text{ commutative}) \tag{35}$$

5 Composing Behaviours

To be useful in the design of large, complex systems, the calculus sketched above has to be extended to model the interaction and coordination of software components with different behaviour models. Such different models are, again, described by different monads. This leads to the introduction of a cofibred category of coalgebras Co over a family of interface functors. Exploiting the constructions in this category leads to a reconstruction of the calculus to accommodate components exhibiting different behavioural patterns. Reference [21] introduces the specification and composition of such (heterogeneous) components in the RAISE meta-language (in which both inductive and coinductive specifications can be accommodated).

An advantage of this approach is that the structural aspects of each behaviour model (for example, inactive vs active, deterministic vs non deterministic, total vs partial, etc.), described by the component signatures, are separated clearly from the interaction structure which defines the interaction and synchronisation among systems (or subsystems, components) and is described by the operations in the category of coalgebras.

The category Co is defined as the 'total' category which encompasses the categories Cp_{B} for all the possible monads B, entailing a notion of morphism between coalgebras for different monads. Recall that a morphism between two monads $\mathsf{B}_1 = (B_1, \eta_1, \mu_1)$ and $\mathsf{B}_2 = (B_2, \eta_2, \mu_2)$ on the category Set is a natural transformation $\alpha : \mathsf{B}_1 \to \mathsf{B}_2$ making the following diagrams to commute:

$$\begin{tikzcd} & \mathbf{Id} \arrow[dl, "\eta_1"'] \arrow[dr, "\eta_2"] & \\ B_1 \arrow[rr, "\alpha"'] & & B_2 \end{tikzcd} \qquad \begin{tikzcd} B_1B_1 \arrow[r, "\alpha^2"] \arrow[d, "\mu_1"'] & B_2B_2 \arrow[d, "\mu_2"] \\ B_1 \arrow[r, "\alpha_\alpha"] & B_2 \end{tikzcd}$$

The morphism α allows us to view every $\mathsf{T}^{\mathsf{B}_1}$-coalgebra $p : U \to \mathsf{T}^{\mathsf{B}_1}(U)$ as a $\mathsf{T}^{\mathsf{B}_2}$-coalgebra $\alpha_U \circ p : U \to \mathsf{T}^{\mathsf{B}_2}(U)$. To define category Co, take seeded

coalgebras for arbitrary monads as objects and for two such coalgebras $(u_p, a_p : U_p \to \mathsf{T}^{\mathsf{B}_1}(U_p))$ and $(u_q, a_q : u_q \to \mathsf{T}^{\mathsf{B}_2}(U_q))$, take (σ, α) as the arrow between them such that $\sigma u_p = u_q$ and the following diagram commutes:

$$\begin{array}{ccccc} U_p & & \xrightarrow{\quad\sigma\quad} & & U_q \\ \downarrow p & & & & q\downarrow \\ \mathsf{T}^{\mathsf{B}_1}(U_p) & \xrightarrow{\alpha_{U_p}} & \mathsf{T}^{\mathsf{B}_2}(U_p) & \xrightarrow{\mathsf{T}^{\mathsf{B}_2}(\sigma)} & \mathsf{T}^{\mathsf{B}_1}(U_q) \end{array}$$

Composition in this category is as defined above, but for the resulting signature: both functors involved are 'merged' together, into a new pattern. Thus, given components $p = (u_p, \overline{a}_p : U_p \to \mathsf{B_p}(U_p \times K)^I)$ and $q = (u_q, \overline{a}_q : U_q \to \mathsf{B_q}(U_q \times O)^K)$, their sequential composition is given by

$$p; q = (\langle u_p, u_q \rangle \in U, \overline{a}_{p;q} : U \to \mathsf{B}(U \times O)^I)$$

where $U = U_p \times U_q$, as before, and $\mathsf{B} = \mathsf{B}_p\mathsf{B}_q$. Note, however, that the simple composition of the corresponding functors B_p and B_q does not always lead necessarily to new monads. In order to define a new monad it is necessary to verify the existence of a natural transformation $\gamma : \mathsf{B}_q\mathsf{B}_p \to \mathsf{B}_p\mathsf{B}_q$ satisfying a number of (coherence) conditions (see [2] for a detailed discussion). A similar situation arises in the definition of the (corresponding) tensor products.

Reference [22] introduces such operators in detail as well as the family of laws they obey. Curiously they are very close to the one in the base calculus.

6 Conclusions and Future Work

The notion of a component discussed here, and stemming from the context of model oriented specification methods, is characterised by the presence of internal state and by an interaction model which reflects the asymmetric nature of input and output. Since the behaviour semantics of each component can be described by the corresponding (final) coalgebra, the semantics of a larger system can be determined by the operations in the category of coalgebras which give the semantics of the interactions among its components and the coalgebras of the components, that means, the semantics of its syntactic constituents. This approach specifies the *compositionality* and *structural transparency* of systems: a system composed of components and connections among them can be considered as a component again and connected to other components, and thereby the internal state space of the system is hidden again to outside.

The bicategorical setting adopted is in debt to previous work by R. Walters and his collaborators on models for deterministic input-driven systems [12, 13]. However, whereas R. Walters' work deals essentially with deterministic systems, our monadic parametrization allows to focus on the relevant structure of components, factoring out details about the specific behavioural effects that may be produced. The hook and feedback combinators (not discussed here, but see [4, 22]) and tensors are also new. Also close to our modelling approach is [15]

which proposes an axiomatization of what is called a 'notion of a process' in a monoidal category. This work, however, does not cover neither the definition of generic combinators nor the development of an associated calculus.

Our present work is concerned with *refinement*, at both the interface and the behavioural levels, and *tunning* of software components to particular 'use contexts'. This last aspect may become relevant in the context of the second author's work on a unified coalgebraic semantics for the UML [18].

Tunning deals with designs in which it is required that a particular component be 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 distinction is totally absent from model-oriented specification methods, often leading to undesirable over specification. In process calculi, on the other hand, it may be traced back to Milner's distinction between *static* and *dynamic* process connectives, the later being understood as the source of temporal extension. In CCS, 'prefixing' is the typical example of a dynamic connective. Our component algebra lacks such an operator, as we are dealing with concrete coalgebras instead of pure behaviours [3]. Notice, on the other hand, that 'choice', which is also a dynamic operator in process calculi, is treated, at component level, as an aggregation combinator.

The way we envisage to address this is as follows. Suppose one wants to tune the use of a component p. First, a decision on what *using a specification* means has to be made. A simple possibility is just a precedence relation between actions. We believe that such an elementary definition will cover several cases actually arising in practice. Once this has been defined, p suffers a 'state extension'-like operation to incorporate 'historical' information. This may be just a record of the last operation executed, or a complete log file for p. Finally, the *use specification* is provided as context information to the extended p component. In each step such information is confronted to the 'historical' record to validate a possible interaction. The resulting behaviour is computed by a strong anamorphism. Context is directly supported in CHARITY [7] via *strength*, which makes the overall idea easy to implement. Conceptually, note that component *tunning* re-introduces in the calculus (a form of) temporal extension.

Acknowledgements. The work of Sun Meng is partially supported by the National Natural Science Foundation fo China, under grant no. 60273001. Comments by José Oliveira and Bernhard K. Aichernig about research described in this paper are gratefully acknowledged.

References

1. R. C. Backhouse, P. Jansson, J. Jeuring, and L. Meertens. Generic programming: An introduction. In S. D. Swierstra, P. R. Henriques, and J. N. Oliveira, editors, *Third International Summer School on Advanced Functional Programming, Braga*, pages 28–115. Springer Lect. Notes Comp. Sci. (1608), September 1998.

2. L. S. Barbosa. *Components as Coalgebras.* PhD thesis, DI, Universidade do Minho, 2001.
3. L. S. Barbosa and J. N. Oliveira. Coinductive interpreters for process calculi. In *Proc. of FLOPS'02*, pages 183–197, Aizu, Japan, September 2002. Springer Lect. Notes Comp. Sci. (2441).
4. L. S. Barbosa and J. N. Oliveira. State-based components made generic. In H. P. Gumm, editor, *Elect. Notes in Theor. Comp. Sci. (CMCS'03 - Workshop on Coalgebraic Methods in Computer Science)*, volume 82.1, Warsaw, April 2003.
5. J. Benabou. Introduction to bicategories. *Springer Lect. Notes Maths. (47)*, pages 1–77, 1967.
6. R. Bird and O. Moor. *The Algebra of Programming.* Series in Computer Science. Prentice-Hall International, 1997.
7. R. Cockett and T. Fukushima. About Charity. Yellow Series Report No. 92/480/18, Dep. Computer Science, University of Calgary, June 1992.
8. J. Goguen, J. Thatcher, and E. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In R. Yeh, editor, *Current Trends in Programming Methodology*, pages 80–149. Prentice-Hall International, 1978.
9. J. Goguen, J. Thatcher, E. Wagner, and J. Wright. Initial algebra semantics and continuous algebras. *Jour. of the ACM*, 24(1):68–95, January 1977.
10. J. Jeuring and P. Jansson. Polytypic programming. In T. Launchbury, E. Meijer, and T. Sheard, editors, *International Summer School on Advanced Functional Programming*, pages 68–114. Springer Lect. Notes Comp. Sci. (1129), 1996.
11. C. B. Jones. *Systematic Software Development Using* VDM. Series in Computer Science. Prentice-Hall International, 1986.
12. P. Katis, N. Sabadini, and R. F. C. Walters. Bicategories of processes. *Journal of Pure and Applied Algebra*, 115(2):141–178, 1997.
13. P. Katis, N. Sabadini, and R. F. C. Walters. On the algebra of systems with feedback and boundary. *Rendiconti del Circolo Matematico di Palermo*, II(63):123–156, 2000.
14. A. Kock. Strong functors and monoidal monads. *Archiv für Mathematik*, 23:113–120, 1972.
15. S. Krstic, J. Launchbury, and D. Pavlovic. Categories of processes enriched in final coalgebras. In *Proceedings of* FOSSACS, pages 303–317. Springer Lect. Notes Comp. Sci. (2030), 2001.
16. R. Milner. *Communication and Concurrency.* Series in Computer Science. Prentice-Hall International, 1989.
17. J. C. Reynolds. Types, abstraction and parametric polymorphism. *Information Processing 83*, pages 513–523, 1983.
18. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley, 1997.
19. J. Rutten. Universal coalgebra: A theory of systems. *Theor. Comp. Sci.*, 249(1):3–80, 2000. (Revised version of CWI Techn. Rep. CS-R9652, 1996).
20. J. M. Spivey. *The Z Notation: A Reference Manual (2nd ed).* Series in Computer Science. Prentice-Hall International, 1992.
21. M. Sun and B. Aichernig. Component-based Coalgebraic Specification and Verification in RSL. Technical Report 267, UNU/IIST, October 2002.
22. M. Sun and B. Aichernig. $\mathbf{Coalg}_{KPF}$: Towards a Coalgebraic Calculus for Component-based Systems. Technical Report 271, UNU/IIST, January 2003.
23. C. Szyperski. *Component Software, Beyond Object-Oriented Programming.* Addison-Wesley, 1998.

24. The RAISE Language Group. *The RAISE Specification Language*. Prentice Hall International, 1992.
25. The RAISE Method Group. *The RAISE Development Method*. Prentice Hall International, 1995.
26. P. Wadler and K. Weihe. Component-based programming under different paradigms. Technical report, Report on the Dagstuhl Seminar 99081, February 1999.