A Perspective on Component Refinement

Luís S. Barbosa[1]

Universidade do Minho, DI - CCTC, Campus de Gualtar,
4710-057 Braga, Portugal
lsb@di.uminho.pt

Abstract. This paper provides an overview of an approach to coalgebraic modelling and refinement of state-based software components, summing up some basic results and introducing a discussion on the interplay between behavioural and classical data refinement. The approach builds on coalgebra theory as a suitable tool to capture observational semantics and to base an abstract characterisation of possible behaviour models for components (from partiality to different degrees of non-determinism).

1 Introduction

In recent years *component-based software development* [46, 49] emerged as a promising paradigm to deal with the ever increasing need for mastering complexity in software design, evolution and reuse. However, as it happened before with object-orientation, and software engineering in the broad sense, component-orientation has grown up to a collection of popular technologies, methods and tools, before consensual definitions and principles (let alone formal foundations) have been put forward.

This paper focus on a particular corner of the 'componentware' landscape. A corner in which software components are regarded as specifications of *state-based* modules, in the tradition of the so-called *model oriented* approach to formal systems design — a widespread paradigm of which VDM [23], Z [45], B [1] and RAISE [47] are well-known representatives. In a series of papers, starting with [6] and including [8, 7, 9], a coalgebraic characterisation of this sort of components and a corresponding calculus was proposed. This approach defines components as persistent units which encapsulate a number of services through public interfaces and provide limited access to internal state spaces. Coalgebra theory [42] was found a suitable tool to capture observational semantics and to base an abstract characterisation of possible *behaviour models* for components (*e.g.*, partiality or (different degrees of) non-determinism). Such models are introduced in the framework in a *generic* [5] way — *i.e.*, as a *parameter*, in the form of a strong monad in the component calculus. More recently in [29, 30] the framework was extended from an equivalence to a refinement calculus, based on a weak form of coalgebra morphism.

This paper provides an overview of this approach to component modelling and refinement, summing up some basic results and introducing a discussion on the interplay between behavioural refinement, as introduced in [29, 30], and classical *data* refinement [18, 35] applied to software components.

Section 2 motivates the use of coalgebras in component modelling and reviews the component calculus introduced in [8, 7]. This paves the way for a detailed discussion of component refinement at both the *behavioural* and *data* levels in sections 3 and 4, respectively. Finally, section 5 introduces some recent research concerns on this topic.

2 Coalgebraic Models for Software Components

2.1 Coalgebras

One of the most elementary models of a software component, or of any computational process whatsoever, is that of a *function*

$$f : O \longleftarrow I$$

which specifies a deterministic transformation rule between two structures I and O. In a (metaphorical) sense, this may be dubbed as the 'engineer's view' of reality: *here is a recipe (a tool, a technology) to build gnus from gnats.*

Often, however, reality is not so simple. For example, one may know how to produce 'gnus' from 'gnats' but not in all cases. This is expressed by observing the output of f in a more refined context: O is replaced by $O + \mathbf{1}$, where $\mathbf{1}$ denotes the singleton datatype and $+$ is datatype sum or coproduct. Then f is said to be a *partial* function. In other situations one may recognise that there is some environmental (or context) information about 'gnats' that, for some reason, should be hidden from input. It may be the case that such information is too extensive to be supplied to f by its user, or that it is shared by other functions as well. It might also be the case that building gnus would eventually modify the environment, thus influencing latter production of more 'gnus'. For U a denotation of such context information, the signature of f becomes[1]

$$f : (O \times U)^U \longleftarrow I$$

A function computed within a context is often referred to as 'state-based', in the sense the word 'state' has in automaton theory — the internal memory of the automaton which both constraints and is constrained by the execution of actions. In fact, the 'nature' of f as a 'state-based function' is made more explicit by rewriting its signature as

$$f : (O \times U)^I \longleftarrow U$$

This, in turn, may suggest an alternative computational model, which (again in a metaphorical sense) one may dub as the 'natural scientist's view'. Instead of

[1] In the sequel we often adopt the standard mathematical notation B^A for funtional dependency, instead of the equivalent $[A \rightarrow B]$ more familiar in computing.

a recipe to build ‘gnus’ from ‘gnats’, we are left with the awareness that *there exist gnus and gnats and that their evolution can be observed.*

The able ‘natural scientist’ will equip herself with the right ‘lens’ — that is, a tool to observe with, which necessarily entails a particular *shape* for observation. The basic ingredients required to support such an ‘observational’, or ‘state-based’, view of computational processes may be summarised as follows:

a *lens*:	◯⌒◯	(a functor T)
an *observation structure*:	◯⌒◯ universe $\xleftarrow{p}$ universe	(a T-coalgebra)

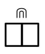
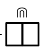
Technically, in the category Set of sets and set-theorectical functions, a *coalgebra* for a functor T is a set U, which corresponds to the object being observed (the *carrier*), and a function $p : \mathsf{T}\,U \longleftarrow U$ [2].

There is, of course, a great diversity of ‘lenses’ and, for the same ‘lens’, a variety of observation structures, *i.e.*, of coalgebras. Moreover, such structures can be related and compared. This entails the need for a notion of *homomorphism*, *i.e.*, a map which preserves the shape of T as an observation tool. Therefore, a T-coalgebra morphism h between, say, coalgebras p and q is just a function between the respective carriers making the following diagram to commute:

$$\begin{tikzcd}
U \arrow[r, "p"] \arrow[d, "h"'] & \mathsf{T}\,U \arrow[d, "\mathsf{T}\,h"] \\
V \arrow[r, "q"'] & \mathsf{T}\,V
\end{tikzcd}$$

Let us consider some possible lenses. An extreme case is the opaque lens: no matter what we try to observe through it, the outcome is always the same. Formally, such a lens is the constant functor $\underline{\mathbf{1}}$ which maps every object to the singleton set $\mathbf{1}$ and every morphism to the identity on $\mathbf{1}$. Since $\mathbf{1}$ is the final object in Set, all $\mathbf{1}$-coalgebras reduce to ! — the canonical function to $\mathbf{1}$. A slightly more interesting lens is $\underline{\mathbf{2}}$, which allows states to be classified into two different classes: black or white. This makes it possible to identify *subsets* of the ‘universe’ U under

[2] The *dual* perspective emphasises the possibility of at least some (essentially finite) things being not only observed, but actually *built*. In this case, one does not work with a ‘lens’ but with a ‘toolbox’. The *assembly process* is specified in a similar (but dual) way:

a *tool box*:		(a functor T)
an *assembly process*:	artifact $\xleftarrow{d}$  artifact	(a T-algebra)

observation, as an observation structure p for this functor maps elements of U to one or another element of $\mathbf{2}$. Should an arbitrary set O be chosen to colour the lens, the possible observations become more discriminating. A coalgebra for $\underline{O}$ is a 'colouring' device in the sense that elements of the universe are classified (*i.e.*, regarded as distinct) by being assigned to different elements of O.

Thus, a 'colour set' as $\mathbf{1}$, $\mathbf{2}$ or O above, can be regarded as a *classifier* of the state space. Coalgebras, for such constant functors, are *pure* observers providing a limited access to the state space by mapping it into the 'colour set' — or *attributes*, as they are known in object-oriented programming.

A common assumption on state-based components is that the state itself is a 'black box': it may evolve either internally or as a reaction to external stimuli, but the only way of tracing such an evolution is by observing the values of its attributes. Under this assumption the 'transparent' lens is not particularly useful. Technically, it corresponds to the *identity* functor Id. An observation structure for Id amounts to a function $p : U \longleftarrow U$. This means that, by using p, the state U can indeed be modified, an ability we hadn't before. But, on the other hand, the absence of attributes makes any meaningful observation impossible. The best we can say, if no direct access to U is allowed, is just that *things happen.* A better alternative is to combine attributes with such state modifiers, or update operations, to model the 'universe' evolution. The latter will be called *actions* here; in the object paradigm they are known as *methods.* Such a combination leads to a richer stock of lens. We might consider, for example, that

- *things happen and disappear*: $\mathsf{T}\,U \;=\; U + \mathbf{1}$
- *things happen and, in doing so, some of their attributes become visible*, *i.e.*, (non trivial) output is produced: $\mathsf{T}\,U \;=\; U \times O$
- additional *input* is required for an observation to take place: $\mathsf{T}\,U \;=\; U^I$
- *we are not completely sure about what has happened*, in the sense that the evolution of the system being observed may be nondeterministic. In this case, the lens above can be combined with $\mathsf{T}\,U \;=\; \mathcal{P}U$ where $\mathcal{P}U$ is the finite powerset of U.

In the second example, the action also has an *input interface.* Typically, actions over the same state space cannot happen simultaneously and, therefore, if more than one is specified in a particular structure, in each execution the input supplied will also select the action to be activated. In some cases, the input is there only for selection purposes: actions with trivial input (*i.e.*, $I = \mathbf{1}$) correspond to buttons that can be pressed. Then the input interface organises itself as a coproduct. Attributes, on the other hand, can be inspected in parallel. In other cases still we might be intereseted in methods which not only change the internal state of a component but also produce an observable output. Putting all the ingredients together we arrive at the following functor as a possible shape for software components modelled as coalgebras:

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

where $A = \Pi_{k \in K} A_k$ stand for the product of the attribute types and $I = \sum_{j \in J} I_j$ and $O = \sum_{j \in J} O_j$ correspond to the coproduct of, respectively, input and output parameters of the component operations.

Functor T can still be enriched with a specification of a particular *behavioural model* to which components may stick to. Notice the use of the *maybe* or the *powerset* monads above to capture such models. Therefore functor T in (1) becomes parametric on an arbitrary *strong monad*[3] B, leading to coalgebras for

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

as a possible general model for state based software components. Therefore 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. Unit (η) and multiplication (μ), provide, respectively, a value embedding and a 'flatten' operation to reduce nested behavioural effects. Strength, either in its right (τ_r) or left (τ_l) version, cater for context information.

Several possibilities can be considered for B. The simplest case is, obviously, the *identity* monad, Id, whereby components behave in a totally *deterministic* way. Other possibilities capturing more complex behavioural features, include the maybe monad ($\mathsf{B} = \mathsf{Id} + \mathbf{1}$) for *partiality*, the (finite) powerset ($\mathsf{B} = \mathcal{P}$) or sequence ($\mathsf{B} = \mathsf{Id}^*$) monads for (arbitrary or ordered) *non determinism* or the *bag monad* [4] to model cases in which among the possible future evolutions of a component, some are stipulated to be more likely (cheaper, more secure, *etc.*) than others (see [8] for further details on the use of monads in a calculus of generic behaviour models).

2.2 Components

Building on the discussion above, this subsection introduces component's specifications as coalgebras and gives a glimpse of the resulting calculus. Without a major loss of generality, however, we shall concentrate in this text on coalgebras for

$$\mathsf{T} = \mathsf{B}(\mathsf{Id} \times O)^I \tag{3}$$

[3] 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 [26]. B being strong means there exist natural transformations $\mathsf{T}(\mathsf{Id} \times -) : \mathsf{T} \times - \Longleftarrow \mathsf{T} \times -$ and $\mathsf{T}(- \times \mathsf{Id}) : - \times \mathsf{T} \Longleftarrow - \times \mathsf{T}$, 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 $\mathsf{B}\mathsf{B}(I \times J)$, 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 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.

[4] defined over a structure $\langle M, \oplus, \otimes \rangle$, where both $\oplus$ and $\otimes$ are Abelian monoids, the latter distributing over the former.

therefore ommiting the attribute's part in (2). Notice that, for $\mathsf{B} = \mathsf{Id}$, such coalgebras correspond to classical *Mealy* machines [27]. In general a *component specification* is defined as follows, where a collection of sets I, O, ..., acting as component interfaces is assumed.

Definition 1. *A software component is specified by a pointed coalgebra*

$$\langle u_p \in U_p, \overline{a}_p : \mathsf{B}(U_p \times O)^I \longleftarrow U_p \rangle \tag{4}$$

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

An elementary, but typical, example of a state based component is given by the following specification of a buffering device which provides services to store and deliver messages.

Example 1 *Denoting by U its internal state, a buffer for messages of type M is handled through operations*

$$\mathsf{put} : U \longleftarrow U \times M$$
$$\mathsf{pick} : U \times M \longleftarrow U$$

An alternative, 'black box' view hides U from the component's environment and regards each operation as a pair of input/output ports. Such a 'port' signature of, e.g., *the* pick *operation is given by*

$$\mathsf{pick} : M \longleftarrow \mathbf{1}$$

The intuition is that pick *is activated with the simple pushing of a 'button' (its argument being the buffer private state space) whose effect is the production of a M value in the corresponding output port. Similarly typing* put *as*

$$\mathsf{put} : \mathbf{1} \longleftarrow M$$

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. Note how input (respectively, output) 'ports' are represented by the sum of their parameters. Such sums label the buffer input (respectively, output) point represented by an empty (respectively, full) circle in the diagram. Combined input type $M + \mathbf{1}$ models the choice between the two functionalities.
One might capture Store *dynamics by a function $a_{\mathsf{Store}} : \mathcal{P}(U \times O) \longleftarrow U \times I$ which describes how* Store *reacts to input stimuli, produces output data (if any) and changes state. This can also be written in a curried form as $\overline{a}_{\mathsf{Store}} : \mathcal{P}(U \times O)^I \longleftarrow U$ that is, as a* coalgebra *of signature $U \longleftarrow \mathsf{T}\, U$ where functor T captures transition 'shape':*

$$\mathsf{T}\ = \mathcal{P}(\mathsf{Id} \times O)^I \tag{5}$$

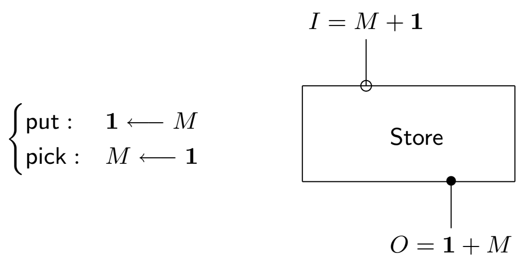

$I = M + \mathbf{1}$
$\begin{cases}\mathsf{put}: & \mathbf{1} \longleftarrow M\\ \mathsf{pick}: & M \longleftarrow \mathbf{1}\end{cases}$
Store
$O = \mathbf{1} + M$

Fig. 1. The Store component.

Built in this 'shape' is the possibility of non deterministic evolution captured by the finite powerset monad. Concretely, our first model for Store *given below assumes that messages are labelled by a time tag (provided by a clock function* ttag()*) so that on the arrival of a* pick *request any message stored for more than a specified time interval (ϵ) can be delivered. Let $U = \mathcal{P}(M \times T)$, where T stands for a suitable representation of time, be its state space. Then,*

$$\begin{aligned} a_{\mathsf{Store}}\langle u, \mathsf{put}\ m\rangle &= \{\langle u \cup \{\langle m, \mathsf{ttag}()\rangle\}, \iota_1 *\rangle\} \\ a_{\mathsf{Store}}\langle u, \mathsf{pick}\rangle &= \{\langle u \setminus \{\langle m, t\rangle\}, \iota_2\ m\rangle \mid \langle m, t\rangle \in u \wedge t - \mathsf{ttag}() \geq \epsilon\} \end{aligned}$$

where put m *and* pick *abbreviate $\iota_1\ m$ and $\iota_2\ *$, respectively.*

Components can be regarded as arrows between (input/output) interfaces and therefore arrows between components are arrows between arrows. Formally, these three notions — *interfaces*, *components* and *component morphisms* — lead to the notion of a *bicategory* [5] as a possible mathematical universe for components to live. In brief, we take interfaces as *objects* of a bicategory Cp, whose *arrows* are pointed T-coalgebras and *2-cells*, the arrows between arrows, the corresponding morphisms. Formally,

Definition 2. *Assume arbitrary sets as* Cp *objects. For each pair $\langle I, O\rangle$ of objects, define a category* Cp(I, O)*, whose arrows*

$$h : \langle u_q, \overline{a}_q\rangle \longleftarrow \langle u_p, \overline{a}_p\rangle \qquad \begin{tikzcd} U_p \arrow[r, "\overline{a}_p"] \arrow[d, "h"'] & \mathsf{T}\, U_p \arrow[d, "\mathsf{T}^{\mathsf{B}}\, h"] \\ U_q \arrow[r, "\overline{a}_q"'] & \mathsf{T}\, U_q \end{tikzcd}$$

[5] Basically a *bicategory* [11] is a category in which a notion of arrows between arrows is additionally considered. This means that the the space of morphisms between any given pair of objects, usually referred to as a (hom-)set, acquires itself the structure of a category. Therefore arrow composition and unit laws become functorial, since they transform both objects and arrows of each hom-set in an uniform way.

satisfy the following morphism *and* seed preservation *conditions:*

$$\overline{a}_q \cdot h \;=\; \mathsf{T}\, h \cdot \overline{a}_p \tag{6}$$

$$h\, u_p \;=\; u_q \tag{7}$$

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

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

whose action on 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 $a_{p;q} : \mathsf{B}(U_p \times U_q \times O) \longleftarrow U_p \times U_q \times I$ *is detailed as follows*

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

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

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

whose action on objects is the constant component $\langle * \in \mathbf{1}, \overline{a}_{\mathsf{copy}_K}\rangle$*, where* $a_{\mathsf{copy}_K} = \eta_{\mathbf{1}\times K}$*. Slightly abusing notation, this will be also referred to as* copy_K*. Similarly, the action on morphisms is the constant morphism* $\mathsf{id}_{\mathbf{1}}$*.*

2.3 A Component Calculus

The fact that, for each strong monad B, components form a bicategory amounts not only to a standard definition of two basic combinators $;$ and copy_K of a possible component calculus, but also to setting up its laws in the form of bisimulation equations. 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:

$$\mathsf{copy}_I\,; p \;\sim\; p \;\sim\; p\,; \mathsf{copy}_O \tag{8}$$

$$(p\,;q)\,;r \;\sim\; p\,;(q\,;r) \tag{9}$$

In previous papers [8, 7, 9] we have proposed an algebra of T-components parametric on a behaviour model B. The development of such a calculus starts from the simple observation that functions can be regarded as particular instances of components, whose interfaces are given by their domain and codomain types. Formally,

Definition 3. *A function $f : B \longleftarrow A$ is represented in* Cp *by*

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

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

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

The pre- and post-composition of a component with Cp-lifted functions can be encapsulated into an unique combinator, called *wrapping*, which is reminiscent of the *renaming* connective found in process calculi (*e.g.*, [31]). Let $p : O \longleftarrow I$ be a component and consider functions $f : I \longleftarrow I'$ and $g : O' \longleftarrow O$. Component p wrapped by f and g, denoted by $p[f, g]$ and typed as $O' \longleftarrow I'$, is defined by input pre-composition with f and output post-composition with g. Formally,

Definition 4. *The* wrapping *combinator is a functor*

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

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

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

Three tensor products are also introduced in the calculus to model *choice* ($\boxplus$), *concurrent* ($\circledast$) and *parallel* composition ($\boxtimes$). The latter is detailed below for illustration purposes.

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. The behavioural effect, captured by monad B, propagates. For example, if B expresses component failure and one of the arguments fails, product fails as well. Formally,

Definition 5. *The* parallel *combinator $\boxtimes$ is defined as $I \boxtimes J \;=\; I \times J$ on objects and a family of functors*

$$\boxtimes_{IOJR} : \mathsf{Cp}(I \times J, O \times R) \longleftarrow \mathsf{Cp}(I, O) \times \mathsf{Cp}(J, R)$$

which yields

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

where

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

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

A generic form of component interaction is achieved by a generalization of sequential composition, leading to a family of *hook* combinators which forces *part* of the output of a component to be fed back as input. For components with the same input/output type, the *hook* combinator has a particularly simple definition as the Kleisli composition of the original coalgebra. Formally,

Definition 6. *Let $p : Z \longleftarrow Z$. Define*

$$p \,\Lsh : Z \longleftarrow Z \;= \langle u_p \in U_p, \overline{a}_{p\Lsh} \rangle$$

where

$$a_{p\Lsh} \;= \qquad U_p \times Z \xrightarrow{a_p} \mathsf{B}(U_p \times Z) \xrightarrow{\mathsf{B}a_p} \mathsf{BB}(U_p \times Z) \xrightarrow{\mu} \mathsf{B}(U_p \times Z)$$

i.e., $a_{p\Lsh} = a_p \bullet a_p$,

The following example illustrates the use of some component combinators to connect elementary state-based specifications. The component to be built is known as the *game of life*, a simple model of cellular behaviour which has been popularised as a common screen locker for computers.

Example 2 *The game is based on a grid of cells each of which sends and receives elementary stimulus to and from its four adjacent neighbours. A stimulus is a Boolean value indicating whether the cell is either 'alive' or 'dead'. The following few rules govern the survival, death and birth of cell generations:*

- *Each living cell with less than two or more than three living neighbours dies in the next generation.*
- *Each dead cell with exactly three living neighbours becomes alive.*
- *Each living cell with less than two or three living neighbours survives until the next generation.*

Each cell will be specified as a component Cell *whose input is a tuple of four Boolean values, each one to be supplied by one of the four adjacent cells. The cell reacts to such a stimulus by computing its new state — 'dead' or 'alive' — and by making it available as an output to its neighbours, used to compute the next cell generation. Formally, we define*

$$\mathsf{Cell} : \mathbf{2} \longleftarrow \mathbf{2} \times \mathbf{2} \times \mathbf{2} \times \mathbf{2} \quad = \quad \langle \mathsf{true} \in \mathbf{2}, \overline{a}_{\mathsf{Cell}} \rangle$$

where

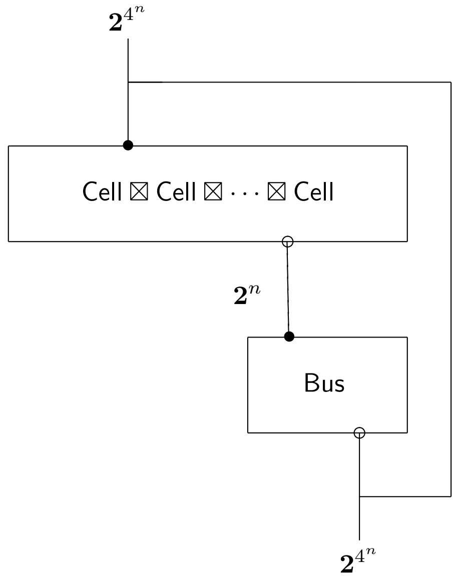
$$a_{\mathsf{Cell}}\ \langle u, t \rangle \;=\; \text{let } n = \mathsf{living}\, t \\ \text{in} \begin{cases} \langle \mathsf{false}, \mathsf{false} \rangle & \text{if } u = \mathsf{true} \wedge (n < 1 \vee n > 3) \\ \langle \mathsf{true}, \mathsf{true} \rangle & \text{if } u = \mathsf{false} \wedge n = 3) \\ \langle u, u \rangle & \text{otherwise} \end{cases}$$

Function living *above, counts the number of living stimuli (*i.e., *the number of* true *values) in a four Boolean tuple. So,* $U_{\mathsf{Cell}} = \mathbf{2}$ *and* $\mathsf{B} = \mathsf{Id}$*. The game's* *behaviour is, of course, deterministic and all cells in the grid react simultaneously to produce the new generation. To form a grid of n cells we simply connect them using the* parallel *combinator $\boxtimes$. The crucial point is to devise a wiring scheme to guarantee that the joint output of the n connected cells is appropriately fed back. The composed system is pictured below, where component*

$$\mathsf{Bus} : \mathbf{2}^{4^n} \longleftarrow \mathbf{2}^n$$

concentrates and correctly distributes the output.

*The n cells are organised as a fully connected matrix of k rows and l columns ($n = k \times l$), so that the neighbours of cell $\langle i, j\rangle$ are $\langle i-1, j\rangle$, $\langle i+1, j\rangle$, $\langle i, j-1\rangle$ and $\langle i, j+1\rangle$ (in the 'west', 'east', 'north' and 'south' directions, respectively) computed in the k and l rings (*i.e., *$1-1 = k$, $k+1 = 1$ and $1-1 = l$, $l+1 = 1$).*


$\mathbf{2}^{4^n}$
$\mathsf{Cell} \boxtimes \mathsf{Cell} \boxtimes \cdots \boxtimes \mathsf{Cell}$
$\mathbf{2}^n$
Bus
$\mathbf{2}^{4^n}$

To specify Bus *we adopt the following convention: the first cell in the $\boxtimes$-expression has coordinates $\langle 1, 1\rangle$, second is $\langle 1, 2\rangle$ and so on until column n is reached; the next cell is then $\langle 2, 1\rangle$. Under this convention the output produced by cell $\langle i, j\rangle$ is selected from the global output tuple as the $j + (n \times (i-1))$-projection,* i.e.

$$\mathsf{out}_{\langle i,j\rangle} : \mathbf{2}^n \longrightarrow \mathbf{2}$$
$$\mathsf{out}_{\langle i,j\rangle} = \pi_{j+(n\times(i-1))}$$

Now, the input to cell $\langle i, j\rangle$ is simply the split *of the outputs of its neighbours,* i.e.,

$$\mathsf{in}_{\langle i,j\rangle} : \mathbf{2}^n \longrightarrow \mathbf{2}^4$$
$$\mathsf{in}_{\langle i,j\rangle} = \langle \mathsf{out}_{\langle i,\mathsf{dec}_n j\rangle}, \mathsf{out}_{\langle \mathsf{dec}_n i,j\rangle}, \mathsf{out}_{\langle i,\mathsf{inc}_n j\rangle}, \mathsf{out}_{\langle \mathsf{inc}_n i,j\rangle} \rangle$$

where $\mathsf{dec}_n x = (x = 1 \rightarrow n,\, x - 1)$ *and* $\mathsf{inc}_n x = (x = n \rightarrow 1,\, x + 1)$. *Finally,* Bus *is defined as the lifting of the* split

$$\mathsf{w} \;=\; \langle \mathsf{in}_{\langle i,j \rangle} \mid i, j \in 1..n \rangle$$

The game of life *component is then written as*

$$\mathsf{GameLife} \;=\; ((\mathsf{Cell} \boxtimes \mathsf{Cell} \boxtimes \cdots \boxtimes \mathsf{Cell}) \,;\, \mathsf{Bus}) \;\Lsh$$

where

$$\mathsf{Bus} \;=\; \ulcorner \mathsf{w} \urcorner$$

Note how the hook *combinator is responsible for extending the game's behaviour to the infinite, once the component has been stimulated with an initial input.*

3 Behavioural Refinement

3.1 Component's Behaviour and Bisimulation

Successive observations of (or experiments with) a T-coalgebra p reveals its behavioural patterns. These are defined in terms of the results of the observers as recorded in the shape T. Then, just as the initial algebra is canonnically defined over the terms generated by successive application of constructors, it is also possible to define a canonical coalgebra in terms of such 'pure' observations. Such a coalgebra is the final object in C_T, if it exists, and will be denoted by out_T over carrier ν_T.

Being *final* means that there exists a *unique* morphism to out_T from each other coalgebra $\langle U, p \rangle$. This is called the *coinductive extension* of p [48] or the *anamorphism* generated by p [28], and written as $[\!(p)\!]_\mathsf{T}$ or, simply, $[\!(p)\!]$, if the functor is clear from context. In other words, an anamorphism is defined as the unique function making the following diagram to commute:

$$\begin{tikzcd}
\nu_\mathsf{T} \arrow[r, "\mathsf{out}_\mathsf{T}"] & \mathsf{T}\,\nu_\mathsf{T} \\
U \arrow[u, "{[\!(p)\!]_\mathsf{T}}"] \arrow[r, "p"] & \mathsf{T}\,U \arrow[u, "{\mathsf{T}\,[\!(p)\!]_\mathsf{T}}"']
\end{tikzcd}$$

or, alternatively, by the following universal law:

$$k = [\!(p)\!]_\mathsf{T} \;\;\Leftrightarrow\;\; \mathsf{out}_\mathsf{T} \cdot k = \mathsf{T}\,k \cdot p \tag{10}$$

For each $u \in U$, $[\!(p)\!]_\mathsf{T}\; u$ can be thought of as the (observable) behaviour of a sequence of p transitions starting at state u. This explains yet another alternative designation for an anamorphism: *unfold* [14]. On its turn, u in $[\!(p)\!]_\mathsf{T}\; u$, is called the *seed* of the anamorphism.

As in the algebraic case, the *existence* part of the universal property (*i.e.*, the implication from left to right) provides a *definition* principle for (circular) functions to the final coalgebra which amounts to equip their source with a coalgebraic structure specifying the 'one-step' dynamics. Then the corresponding anamorphism gives the rest. In other words, such functions are defined by specifying their output under all different observers. The *uniqueness* part (*i.e.*, the reverse implication), on the other hand, offers a powerful *proof* principle — *coinduction.*

Agreeing with the intuition that the final coalgebra is the coalgebra of *all* behaviours, *observational equivalence* can be defined as

$$u \sim v \iff [\![p]\!]\ u = [\![q]\!]\ v \tag{11}$$

for u and v in the carriers of coalgebras $\langle U, p\rangle$ and $\langle V, q\rangle$, respectively. The notion of a *bisimulation*, which is central in coalgebra theory [42], entails a *local* proof theory for observational equivalence. Informally, two states of a T-coalgebra (or of two different T-coalgebras) are related by a bisimulation if their observation produces equal results and this is maintained along all possible transitions. I. e., each one can mimic the other's evolution. Originally the notion of bisimulation, which can be traced back to [44] and [12], was introduced in the context of process calculi in Park's landmark paper [38]. Later [2] gave a categorical definition which applies, not only to the kind of transition systems underlying the operational semantics of process calculi, but also to arbitrary coalgebras. Bisimulation *acquired a shape*: the shape of the chosen observation interface T.

A notion of *refinement* should also be shaped by T. Intuitively component p is a *behavioural refinement* of q if the behaviour patterns observed from p are a *structural restriction*, with respect to the *behavioural model* captured by monad B, of those of q. To make precise such a 'definition' we shall first describe behaviour patterns concretely as *generalized transitions.*

3.2 Refinement

Just as transition systems can be coded back as coalgebras, any coalgebra $\langle U, \alpha : \mathsf{T}U \longleftarrow U\rangle$ specifies a (T-shaped) transition structure over its carrier U. For extended polynomial Set endofunctors[6] such a structure may be expressed as a binary relation ${}_{\alpha}\!\longleftarrow : U \longleftarrow U$, defined in terms of the *structural membership* relation (which is an instance of generic datatype membership [19]) $\in_{\mathsf{T}} : U \longleftarrow \mathsf{T}\, U$, *i.e.*,

$$u' \;{}_{\alpha}\!\longleftarrow u \;\equiv\; u' \in_{\mathsf{T}} \alpha\, u$$

or, in an equivalent but pointfree formulation which often simplifies formal reasoning, as the following relational equality[7]

[6] The class inductively defined as the least collection of functors containing the identity Id and constant functors K for all objects K in the category, closed by functor composition and finite application of product, coproduct, covariant exponential and finite powerset functors.

[7] In the sequel both functional and relational composition will be denoted by the same symbol $\cdot$ given that the former is just a special case of the latter.

$$\underset{\alpha}{\longleftarrow} \;=\; \in_{\mathsf{T}} \cdot \alpha$$

where $\in_{\mathsf{T}}$ is defined by induction on the structure of T:

$$\begin{array}{rcl}
x \in_{\mathsf{Id}} y & \text{iff} & x = y \\
x \in_{K} y & \text{iff} & \mathsf{false} \\
x \in_{\mathsf{T}_1 \times \mathsf{T}_2} y & \text{iff} & x \in_{\mathsf{T}_1} \pi_1\, y \;\vee\; x \in_{\mathsf{T}_2} \pi_2\, y \\
x \in_{\mathsf{T}_1 + \mathsf{T}_2} y & \text{iff} & \begin{cases} y = \iota_1\, y' & \Rightarrow x \in_{\mathsf{T}_1} y' \\ y = \iota_2\, y' & \Rightarrow x \in_{\mathsf{T}_2} y' \end{cases} \\
x \in_{\mathsf{T}^K} y & \text{iff} & \exists_{k \in K}.\; x \in_{\mathsf{T}} y\, k \\
x \in_{\mathcal{P}\mathsf{T}} y & \text{iff} & \exists_{y' \in y}.\; x \in_{\mathsf{T}} y'
\end{array}$$

For any function h, relation $\in_{\mathsf{T}}$ satisfies the following naturality condition

$$h \cdot \in_{\mathsf{T}} \;=\; \in_{\mathsf{T}} \cdot \mathsf{T}\, h \tag{12}$$

which can be proved by induction on T. Applying *shunting*[8] to the left to rigth inclusion component of equation (12) leads to

$$\in_{\mathsf{T}} \;\subseteq\; h^{\circ} \cdot \in_{\mathsf{T}} \cdot \mathsf{T}h \tag{13}$$

The dynamics of a component $p : O \longleftarrow I$ is based on functor $\mathsf{B}(\mathsf{Id} \times O)^I$. Therefore a possible (and intuitive) way of regarding component p as a behavioural refinement of q is to consider that p transitions are simply *preserved* in q. For non deterministic components this is understood as set inclusion. But one may also want to consider additional restrictions. For example, to stipulate that if p has no transitions from a given state, q should also have no transitions from the corresponding state(s). In any case the basic question is: how can such a refinement situation be identified?

The general 'recipe' to identify a refinement situation is to look for an *abstraction* to witness it [18]. In other words: look for a morphism in the relevant category, from the 'concrete' to the 'abstract' model such that the latter can be *recovered* from the former up to a suitable notion of equivalence, though, typically, not in a unique way. Component morphisms, however, are (seed preserving) coalgebra morphisms which are known to entail bisimilarity. Actually a T-coalgebra morphism $h : \beta \longleftarrow \alpha$ is a function from the state space of α to that of β such that

$$\mathsf{T}h \cdot \alpha \;=\; \beta \cdot h \tag{14}$$

Regarding α and β as (generalised) transition systems equation (14) becomes a relational equality:

$$h \cdot \underset{\alpha}{\longleftarrow} \;=\; \underset{\beta}{\longleftarrow} \cdot h \tag{15}$$

[8] In the relational calculus [4] Galois connection $f \cdot R \subseteq S \;\equiv\; R \subseteq f^{\circ} \cdot S$, involving function f and arbitrary relations R and S, is known as the *shunting rule*. Also note that notation R° stands for the *converse* of relation R.

i.e., the conjunction of inclusions

$$h \cdot \ {}_{\alpha}{\longleftarrow} \ \subseteq \ {}_{\beta}{\longleftarrow} \cdot h \tag{16}$$

$${}_{\beta}{\longleftarrow} \cdot h \ \subseteq \ h \cdot \ {}_{\alpha}{\longleftarrow} \tag{17}$$

which, by introducing variables and observing that by *shunting* inclusion (16) can also be presented as ${}_{\alpha}{\longleftarrow} \subseteq h^{\circ} \cdot {}_{\beta}{\longleftarrow} \cdot h$, takes the following more familiar shape

$$u' \ {}_{\alpha}{\longleftarrow} \ u \ \Rightarrow \ h\, u' \ {}_{\beta}{\longleftarrow} \ h\, u \tag{18}$$

$$v' \ {}_{\beta}{\longleftarrow} \ h\, u \ \Rightarrow \ \exists_{u' \in U}.\ u' \ {}_{\alpha}{\longleftarrow} \ u \ \wedge \ u' = h\, v' \tag{19}$$

They jointly state that, not only α dynamics, as represented by the induced transition relation, is *preserved* by h (16), but also β dynamics is *reflected* back over the same h (17). Is it possible to weaken the morphism definition to capture only one of these aspects? In [29] this question got an afirmative answer, resorting to the notion of a preorder $\leq$ on a Set endofunctor T introduced in [21]. Such a preorder is defined as a functor $\leq$ in such a way that, for any function $h : V \longleftarrow U$, $\mathsf{T}h$ preserves the order, *i.e.*

$$x_1 \leq_{\mathsf{T}X} x_2 \ \Rightarrow \ (\mathsf{T}h)\, x_1 \leq_{\mathsf{T}Y} (\mathsf{T}h)\, x_2 \tag{20}$$

or, in a pointfree formulation,

$$(\mathsf{T}h) \cdot \leq \ \subseteq \ \leq \cdot (\mathsf{T}\, h) \tag{21}$$

Let us denote by $\dot{\leq}$ the pointwise lifting of $\leq$ to the functional level, *i.e.*

$$f \mathrel{\dot{\leq}} g \ \equiv \ \forall_x .\ f\, x \leq g\, x \tag{22}$$

which can also be formulated in the following pointfree way as

$$f \mathrel{\dot{\leq}} g \ \equiv \ f \subseteq \leq \cdot g \tag{23}$$

In [30] it is shown that, for any function h monotonic with respect to $\leq$ one has

$$f \mathrel{\dot{\leq}} g \Rightarrow h \cdot f \mathrel{\dot{\leq}} h \cdot g \tag{24}$$

$$f \mathrel{\dot{\leq}} g \Rightarrow f \cdot h \mathrel{\dot{\leq}} g \cdot h \tag{25}$$

In this context the main result in the above mentioned reference is the definition of a *forward* morphism $h : \beta \longleftarrow \alpha$ with respect to $\leq$ as a function from U to V such that

$$\mathsf{T}\, h \cdot \alpha \ \dot{\leq} \ \beta \cdot h$$

and the proof that

Lemma 1. *For T an endofunctor in Set, T-coalgebras and forward morphisms define a category. Moreover, forward morphisms preserve transitions if $\leq$ is compatible with the membership relation,* i.e., *for all $x \in X$ and $x_1, x_2 \in \mathsf{T}X$,*

$$x \in_{\mathsf{T}} x_1 \;\wedge\; x_1 \leq\; x_2 \;\;\Rightarrow\;\; x \in_{\mathsf{T}} x_2 \tag{26}$$

or, in a pointfree formulation,

$$\in_{\mathsf{T}} \cdot \leq \;\subseteq\; \in_{\mathsf{T}} \tag{27}$$

Proof. We prove only the second part (see [30] for the full proof). Let h be a forward morphism. Transition preservation follows from

$$
\begin{array}{ll}
 & {}_{\alpha}\!\longleftarrow \\
= & \{ \text{ definition } \} \\
 & \in_{\mathsf{T}} \cdot \alpha \\
\subseteq & \{ \text{ (13), monotonicity } \} \\
 & h^{\circ} \cdot \in_{\mathsf{T}} \cdot \mathsf{T}\, h \cdot \alpha \\
\subseteq & \{ \; h \text{ forward entails } \mathsf{T}h \cdot \alpha \subseteq \leq \cdot \beta \cdot h, \text{ monotonicity } \} \\
 & h^{\circ} \cdot \in_{\mathsf{T}} \cdot \leq \cdot \beta \cdot h \\
\subseteq & \{ \text{ compatibility with } \in_{\mathsf{T}} \text{ (27), monotonicity } \} \\
 & h^{\circ} \cdot \in_{\mathsf{T}} \cdot \beta \cdot h \\
= & \{ \text{ definition } \} \\
 & h^{\circ} \cdot {}_{\beta}\!\longleftarrow \cdot h
\end{array}
$$

□

A preorder $\leq$ on an endofunctor T satisfying inclusion (27) will be referred to, in the sequel, as a *refinement preorder*. Then, the existence of a *forward* morphism connecting two components p and q witnesses a refinement situation whose symmetric closure coincides, as expected, with bisimulation. Formally,

Definition 7. *Component p is a behaviour refinement of q, written $q \trianglelefteq p$, if there exist components r and s and a (seed preserving) forward morphism h such that*

$$q \;\sim\; s \xleftarrow{\quad h \quad} r \;\sim\; p$$

The exact meaning of a refinement assertion $q \trianglelefteq p$ depends, of course, on the concrete refinement preorder $\leq$ adopted. But what do we know about such preorders? Condition (27) provides an upper bound leading to a definition of *structural inclusion*:

$$x \subseteq_{\mathsf{Id}} y \;\;\text{ iff }\;\; \forall_{e \in_{\mathsf{T}} x}.\;\; e \in_{\mathsf{T}} y \tag{28}$$

Several other cases arise by suitable restrictions. For example,

- Structural inclusion as defined above is too large to be useful in practice. Actually its definition on a constant functor is the universal relation which would make refinement blind to the outputs produced. This suggests an additional requirement on refinement preorders for Cp components: their definition on a constant functor K must be equality on set K, *i.e.*, $x \leq_K y$ iff $x =_K y$ so that transitions with different O-labels could not be related. Note that refinement of non deterministic components based on this preorder captures the classical notion of *non determinism reduction*.
- A 'failure forcing' variant — $\subseteq^E_{\mathsf{T}}$, where E stands for 'emptyset' — guarantees that the concrete component fails no more than the abstract one. It is defined as $\subseteq_{\mathsf{T}}$ by replacing the clause for the powerset functor by

$$x \subseteq^E_{\mathcal{P}\mathsf{T}} y \quad \text{iff} \quad (x = \emptyset \Rightarrow y = \emptyset) \;\wedge\; \forall_{e \in x} \exists_{e' \in y} .\; e \subseteq_{\mathsf{T}} e'$$

- For partial components refinement based on the preorders above collapse into bisimilarity instead of entailing an increase of definition in the implementation side. An alternative is relation $\subseteq^F_{\mathsf{T}}$ (F standing for 'failure') which adds a *maybe* clause

$$x \subseteq^F_{\mathsf{T}+\mathbf{1}} y \quad \text{iff} \quad \begin{cases} x = \iota_1\, x' \wedge y = \iota_1\, y' & \Rightarrow x' \subseteq_{\mathsf{T}} y' \\ x = \iota_2\, * & \Rightarrow \mathsf{true} \end{cases}$$

  taking precedence over the general sum clause.

We end this section with a small example. The reader is referred to [30] for a glimpse of a refinement calculus for state based components based on the existence of forward morphisms.

Example 3 *For an example of behavioural refinement consider a new specification of component* Store *which differs from the one in example 1 only in the specification of operation* pick. *The idea is that instead of choosing the message to be returned non deterministically from the set of messages waiting for more than a specified* ϵ *delay, the operation selects the message in that set which is waiting for a longer time. Formally, assuming function* lwait *computes such message, the specification becomes*

$$\begin{aligned} a_{\mathsf{Store}}\langle u, \mathsf{pick}\rangle \;=\; & \text{let } c = \{\langle m,t\rangle \in u \mid t - \mathsf{ttag}() \geq \epsilon\} \\ & \text{in } (c = \emptyset \;\rightarrow\; \emptyset,\ \text{let } \langle m,t\rangle = \mathsf{lwait}\, c \text{ in } \{\langle u \setminus \{\langle m,t\rangle\}, \iota_2\, m\rangle\}) \end{aligned}$$

where notation $(\phi \;\rightarrow\; f,\, g)$ *reads* if ϕ then f else g. *Clearly, the identity morphism from this new coalgebra to the one in example 1 is a forward morphism witnessing the former as a refinement of the latter.*

4 Data Refinement

4.1 State Refinement

In the previous section component's refinement was discussed at the *behaviour* level based on a refinement preorder with respect to monad B. There is, however, another axis for refinement: the *data* level, which amounts to the static refinement of the data structure which specifies the component state space. Data refinement, as discussed in formal development methodologies such as VDM [23], is the process of transforming abstract data structures into more concrete ones, a transformation which presumably entails efficiency (*e.g.*, the conversion of inductive data types into 'pointer'-based representations).

Refinement of a component specification p into another specification q has to fulfil a number of requirements. First of all, the existence of *enough redundancy* in the state space of q to represent all the elements in p is required. This is called in [23] the *adequacy* requirement and is captured by the definition of a surjection from the state space of q to that of p, called the *abstraction* or *retrieve* map. Next, *substitution* is regarded as 'complete' in the sense that (concrete) actions in q accept all the input values accepted by the corresponding abstract ones, and, for the same inputs, the results produced are also the same, up to the retrieve map. If components are specified, as they usually are in VDM, by pre and post-conditions, this amounts to say that, under refinement, neither pre-conditions are strengthened, nor post-conditions are weakened. Note this approach to data refinement, which can be traced back to Hoare's landmark paper [18], is usual in *model-oriented* design methods, even though several variants and alternatives have been proposed in the literature (see [41] for a recent account).

In this section we shall resort to SETS [34, 35] — a calculus of data representations, based on identical principles: any refinement is witnessed by a surjection which, whenever partial, may induce a *representation invariant* on the concrete side. Each concrete operation is then *calculated* (rather than 'conjectured and verified' as in VDM) by solving the corresponding refinement diagram.

The calculus consists of inequations of the form $A \leq B$ (read: *data type B refines or implements data type A*) which witnesses the existence of an *abstraction* map abs from B to A with a *right-inverse* rep (called the *representation* relation), *i.e.*,

$$A \leq B \quad \text{iff} \quad \begin{tikzcd} A \arrow[r, bend right, "\mathsf{rep}"'] \arrow[r, phantom, "\leq"] & B \arrow[l, bend right, "\mathsf{abs}"'] \end{tikzcd} \quad \text{such that} \quad \mathsf{abs} \cdot \mathsf{rep} = \mathsf{id}_A \qquad (29)$$

Note that rep is *injective* because $\mathit{ker}\,\mathsf{rep} \subseteq \mathit{ker}\,\mathsf{abs}^\circ$ and $\mathit{ker}\,\mathsf{abs}^\circ = \mathit{img}\,\mathsf{abs}$ which coincides with identity as abs is surjective[9]. Moreover if abs is *partial*, the characteristic predicate of the codomain of relation rep defines the *invariant* induced by the refinement process.

Clearly (see [35] for a proof), the refinement relation is a preorder and is preserved by extended polynomial functors, *i.e.*,

$$\begin{tikzcd} A \arrow[r, bend right, "\mathsf{rep}"'] \arrow[r, phantom, "\leq"] & B \arrow[l, bend right, "\mathsf{abs}"'] \end{tikzcd} \Rightarrow \begin{tikzcd} \mathsf{T}\,A \arrow[r, bend right, "\mathsf{T}\ \mathsf{rep}"'] \arrow[r, phantom, "\leq"] & \mathsf{T}\,B \arrow[l, bend right, "\mathsf{T}\ \mathsf{abs}"'] \end{tikzcd} \qquad (30)$$

[9] Notation $\mathit{ker}\,R$ (respectively, $\mathit{img}\,R$) stand for the kernel (respectively, image) of relation R defined as $\mathit{ker}\,R = R^\circ \cdot R$ (respectively, $\mathit{img}\,R = R \cdot R^\circ$) [4].

Example 4 *A simple example of data refinement in the context of the* Store *component is the implementation of its state space* $U = \mathcal{P}(M \times T)$ *as a finite sequence of type* $(M \times T)^*$ *with* $\mathsf{abs} = \mathsf{elems}$*, the function which returns the set of elements of a list. Other representations for sets, including the notion of a* bag *(which retains the unordered structure of set while keeping track of element repetition) are recorded in the following inequations:*

$$\mathcal{P}A \;\leq\; Nat \leftharpoonup A \;\leq\; A^* \tag{31}$$

where notation $B \leftharpoonup A$ *stands for a partial function (also called a simple relation in [4] or a* mapping *in specification methods like* VDM *[23]) from* A *to* B*. An elementary example of a data refinement situation where the abstraction morphism is not a function is the following representations of elements as 'pointers':*

$$\begin{tikzcd} A \arrow[r, bend right, "\mathsf{rep}=\iota_1"'] \arrow[r, phantom, "\leq"] & A+\mathbf{1} \arrow[l, bend right, "\mathsf{abs}=\iota_1^\circ"'] \end{tikzcd} \tag{32}$$

which, moreover, induce the concrete invariant $\phi = [\mathsf{true}, \mathsf{false}]$ *over the implementation type. References [34, 35] and [36] provide several applications of this calculus to the derivation of imperative programs and data base schemes.*

Once the state space of a component $p = \langle u \in U, \alpha : \mathsf{T}U \longleftarrow U\rangle$ is refined into, say, V a new component is defined over V, whose seed is given by $\mathsf{abs}\, u$ and the dynamics $\beta : \mathsf{T}V \longleftarrow V$ computed as a solution to the following equation

$$\alpha \;=\; \mathsf{T}\mathsf{abs} \cdot \beta \cdot \mathsf{rep} \tag{33}$$

The basic result, from the point of view of a component calculus, is that data refinement entails bisimilarity, *i.e.*,

Lemma 2. *Components* p *and* q *as defined above satisfy* $p \sim q$*.*

Proof. Consider the general case in which refinement $U \leq V$, witnessed by abs and rep, induces a concrete invariant ϕ over V, *i.e.*, an inclusion $i : V \longleftarrow V_\phi$. Let also β' denote the restriction of β to V_ϕ. The starting point is equation (33) which defines the dynamics of q. The target is to show that $\mathsf{abs}_\phi = \mathsf{abs} \cdot i$ is a coalgebra morphism. I.e.,

$$\begin{array}{ll} & \alpha \cdot \mathsf{abs}_\phi \;=\; \mathsf{T}\mathsf{abs}_\phi \cdot \beta' \\ \equiv & \quad \{ \; \mathsf{abs}_\phi = \mathsf{abs} \cdot i \; \} \\ & \alpha \cdot \mathsf{abs} \cdot i \;=\; \mathsf{T}\mathsf{abs} \cdot \mathsf{T}i \cdot \beta' \\ \equiv & \quad \{ \; i \text{ is a coalgebra morphism, } \textit{i.e.}, \; \beta \cdot i = \mathsf{T}i \cdot \beta' \; \} \\ & \alpha \cdot \mathsf{abs} \cdot i \;=\; \mathsf{T}\mathsf{abs} \cdot \beta \cdot i \\ \Rightarrow & \quad \{ \; i \text{ is injective} \; \} \\ & \alpha \cdot \mathsf{abs} \;=\; \mathsf{T}\mathsf{abs} \cdot \beta \end{array}$$

$$\begin{aligned}
\Rightarrow \quad & \{ \text{ Leibniz equality } \} \\
& \alpha \cdot \mathsf{abs} \cdot \mathsf{rep} \;=\; \mathsf{T}\mathsf{abs} \cdot \beta \cdot \mathsf{rep} \\
\equiv \quad & \{ \mathsf{abs} \cdot \mathsf{rep} = \mathsf{id} \text{ and } (33) \} \\
& \alpha \;=\; \alpha
\end{aligned}$$

4.2 Shape Refinement

In the approach to component modelling discussed in this paper, *interfaces* are encoded in the shape of functor T corresponding to the component's service signature. Therefore applying data refinement to this level may capture some forms of interface *enrichment*. Consider, for example, the elementary cases of adding an attribute or an operation to the *shape* of a (deterministic) component:

– adding an *attribute* $\mathsf{at} : B \longleftarrow X$

$$\begin{tikzcd} A \times X \arrow[r, bend right] & (A \times B) \times X \arrow[l, bend right, "\mathsf{abs} = \pi_1 \times \mathsf{id}"'] \end{tikzcd}$$

– adding an *operation* $\mathsf{op} : X \longleftarrow X \times B$

$$\begin{tikzcd} X^A \arrow[r, bend right] & X^{A+B} \arrow[l, bend right, "\mathsf{abs} = (\cdot \iota_1)"'] \end{tikzcd}$$

Now the interesting result is that refinement of the signature functor has a counterpart at the behavioural level, *i.e.*, the carriers of the corresponding final coalgebras, which form the spaces of their behaviours, are also related by a data refinement. Formally, we prove that the data refinement relation as introduced above extends to coinductive types:

Lemma 3. *Let* T *and* G *be extended polynomial functors. Then,*

$$\begin{tikzcd} \mathsf{T}X \arrow[r, bend right, "\mathsf{rep}"'] \arrow[r, phantom, "\leq"] & \mathsf{G}X \arrow[l, bend right, "\mathsf{abs}"'] \end{tikzcd} \quad \Rightarrow \quad \begin{tikzcd} \nu_{\mathsf{T}} \arrow[r, bend right, "\mathsf{rep}_\nu"'] \arrow[r, phantom, "\leq"] & \nu_{\mathsf{G}} \arrow[l, bend right, "\mathsf{abs}_\nu"'] \end{tikzcd} \tag{34}$$

where ν_{T} *denotes the carrier of the final* T*-coalgebra. Moreover,*

$$\mathsf{abs}_\nu \;\triangleq\; [\![\mathsf{abs} \cdot \mathsf{out}_{\mathsf{G}}]\!]_{\mathsf{T}} \quad \text{and} \quad \mathsf{rep}_\nu \;\triangleq\; [\![\mathsf{rep} \cdot \mathsf{out}_{\mathsf{T}}]\!]_{\mathsf{G}}$$

for abs *and* rep natural *on* X

Proof. We have to show that

$$[\![\mathsf{abs} \cdot \mathsf{out}_{\mathsf{G}}]\!]_{\mathsf{T}} \;\cdot\; [\![\mathsf{rep} \cdot \mathsf{out}_{\mathsf{T}}]\!]_{\mathsf{G}} \;=\; \mathsf{id} \tag{35}$$

in the context of the following diagram:

$$\begin{tikzcd}
\nu_{\mathsf{G}} \arrow[rr, bend left, "\mathsf{out_G}"] \arrow[d, bend right, "(\!|\mathsf{abs} \cdot \mathsf{out_G}|\!)_{\mathsf{T}}"'] & \mathsf{T}\nu_{\mathsf{G}} \arrow[r, bend right, "\mathsf{rep}"'] & \mathsf{G}\nu_{\mathsf{G}} \arrow[l, bend right, "\mathsf{abs}"'] \\
\nu_{\mathsf{T}} \arrow[u, bend right, "(\!|\mathsf{rep} \cdot \mathsf{out_T}|\!)_{\mathsf{G}}"'] \arrow[r, "\mathsf{out_T}"] & \mathsf{T}\nu_{\mathsf{T}} \arrow[r, bend right, "\mathsf{rep}"'] & \mathsf{G}\nu_{\mathsf{T}} \arrow[l, bend right, "\mathsf{abs}"']
\end{tikzcd}$$

Therefore[10],

$$\begin{array}{ll}
& (\!|\mathsf{abs} \cdot \mathsf{out_G}|\!)_{\mathsf{T}} \cdot (\!|\mathsf{rep} \cdot \mathsf{out_T}|\!)_{\mathsf{G}} = \mathsf{id} \\
\equiv & \quad \{ \text{ reflection for coinductive extension } \} \\
& (\!|\mathsf{abs} \cdot \mathsf{out_G}|\!)_{\mathsf{T}} \cdot (\!|\mathsf{rep} \cdot \mathsf{out_T}|\!)_{\mathsf{G}} = (\!|\mathsf{out_T}|\!)_{\mathsf{T}} \\
\Leftarrow & \quad \{ \text{ fusion for coinductive extension } \} \\
& \mathsf{abs} \cdot \mathsf{out_G} \cdot (\!|\mathsf{rep} \cdot \mathsf{out_T}|\!)_{\mathsf{G}} = \mathsf{T}(\!|\mathsf{rep} \cdot \mathsf{out_T}|\!)_{\mathsf{G}} \cdot \mathsf{out_T} \\
\equiv & \quad \{ \text{ cancellation for coinductive extension } \} \\
& \mathsf{abs} \cdot \mathsf{G}(\!|\mathsf{rep} \cdot \mathsf{out_T}|\!)_{\mathsf{G}} \cdot \mathsf{rep} \cdot \mathsf{out_T} = \mathsf{T}(\!|\mathsf{rep} \cdot \mathsf{out_T}|\!)_{\mathsf{G}} \cdot \mathsf{out_T} \\
\equiv & \quad \{ \text{ } \mathsf{rep} \text{ is natural} \} \\
& \mathsf{abs} \cdot \mathsf{rep} \cdot \mathsf{T}(\!|\mathsf{rep} \cdot \mathsf{out_T}|\!)_{\mathsf{G}} \cdot \mathsf{out_T} = \mathsf{T}(\!|\mathsf{rep} \cdot \mathsf{out_T}|\!)_{\mathsf{G}} \cdot \mathsf{out_T} \\
\equiv & \quad \{ \text{ hip } \} \\
& \mathsf{T}(\!|\mathsf{rep} \cdot \mathsf{out_T}|\!)_{\mathsf{G}} \cdot \mathsf{out_T} = \mathsf{T}(\!|\mathsf{rep} \cdot \mathsf{out_T}|\!)_{\mathsf{G}} \cdot \mathsf{out_T}
\end{array}$$

Example 5 *Another typical example is what could be called* stream completion *induced by the following data refinement at the signature level:*

$$\begin{tikzcd}
\mathbf{1} + A \times X \arrow[r, bend right] & (A + \mathbf{1}) \times X \cong X + A \times X \arrow[l, bend right, "\mathsf{abs} = ! + \mathsf{id}"']
\end{tikzcd}$$

Note that the final coalgebra for the 'abstract' shape is A^∞, i.e., *the space of finite and infiinite sequences of* A, *whereas for the concrete case one gets* $(A+\mathbf{1})^\omega$, i.e.,

[10] The laws of *reflection*, *cancellation* and *fusion* stated below, in this order, and used in the proof are standard results on coinduction easily derived from the universal property (10) [15].

$$\begin{array}{rcl}
(\!|\mathsf{out_T}|\!) & = & \mathsf{id}_{\nu_{\mathsf{T}}} \\
\mathsf{out_T} \cdot (\!|p|\!) & = & \mathsf{T}\ (\!|p|\!) \cdot p \\
(\!|p|\!) \cdot h & = & (\!|q|\!) \quad \text{if} \quad p \cdot h \ = \ \mathsf{T}\ h \cdot q
\end{array}$$

streams of either elements of A or a mark $* \in \mathbf{1}$. By the lemma above one may easily conclude that $A^\infty \leq (A+\mathbf{1})^\omega$, a fact often used in coalgebraic specification [22], where a finite sequence is extended to a stream by replication of a dummy value.

5 Conclusions and Further Work

This paper provided an overview of an approach to refinement of (state-based) components whose main theory has been developed in previous publications (namely, [29, 30]). The integration of behaviour and data refinement and the application of the latter to a form of interface enrichement is, however, new.

The main possible interest of this approach is its parametrization by a model of behaviour captured by a strong monad B. This is *generic* enough to capture a number of situations, depending on both B and the refinement preorder adopted. Non determinism reduction is one possibility among many others. For example, Poll's notion of *behavioural subtyping* in [39], at the model level, also emerges as another instantiation.

A note on related work is now in order. First of all two major influences should be acknowledged. The first one relates to the use of a bicategorical setting to capture the 'two-level structure' in component models which is in debt to previous work by R. Walters and his collaborators on models for deterministic input-driven systems [24, 25]. The other is the recent area of coalgebraic specification of object-oriented systems (see *e.g.*, [40, 20]), which has been developed with a similar motivation, although in a property-oriented, or axiomatic, framework.

An alternative, but related, approach to componentware is inspired by research on coordination languages [17, 37] and favors strict component decoupling in order to support a looser inter-component dependency. Here computation and coordination are clearly separated, communication becomes *anonymous* and component interconnection is externally controlled. This model is (partially) implemented in JavaSpaces on top of Jini [33] and fundamental to a number of approaches to componentware which identify communication by generic channels as the basic interaction mechanism — see, *e.g.*, Reo [3], Piccola [43, 32], as well as [16, 13] or [10].

The genericity of the approach described in this paper and its coalgebraic basis seems promising, although a lot of work remains to be done. Among the current research directions we would like to underline the following two.

Backwards refinement. Behavioural refinement was defined in section 3 in terms of transition preservation, *i.e.*, as a sort of T-shaped simulation witnessed by what we have called *forward* morphisms. An alternative point of view is based on the dual notion of *backward* morphisms, morphisms which verify

$$\beta \cdot h \ \dot{\leq}\ \mathsf{T}\, h \cdot \alpha \tag{36}$$

In [29, 30] these are shown to form a category and to *reflect* transitions, in the sense of equation (19), although possible applications to component refinement are still to be developed.

Induced distribution. Several laws of data refinement split components' state space into a number of factors. Typical examples are laws whose purpose is to factorize mappings with structured domains or codomains, heavily used in the derivation of database implementations [36]. For example the following laws, studied in [35], refine a mapping to either a sum or a product type into a product of two mappings:

$$\begin{tikzcd} (B+C) \leftharpoonup A \arrow[rr, bend right] & \leq & (B \leftharpoonup A) \times (C \leftharpoonup A) \arrow[ll, bend right, "\mathsf{cojoin}"'] \end{tikzcd}$$

$$\begin{tikzcd} (B \times C) \leftharpoonup A \arrow[rr, bend right] & \leq & (B \leftharpoonup A) \times (C \leftharpoonup A) \arrow[ll, bend right, "\mathsf{join}"'] \end{tikzcd}$$

where abstractions are defined as

$$\begin{aligned} \mathsf{cojoin} &= \cup \cdot ((\iota_1 \cdot) \times (\iota_2 \cdot)) \\ \mathsf{join} &= \qquad \langle\ ,\ \rangle \end{aligned}$$

where $\langle R, S\rangle = \cap \cdot ((\pi_1^\circ \cdot R) \times (\pi_2^\circ \cdot S))$, is a relational *split* [15]. Similarly relational *either* witnesses the decomposition of a mapping from a sum type:

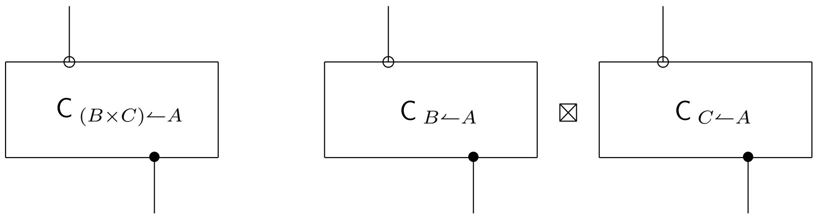
$$\begin{tikzcd} A \leftharpoonup (B+C) \arrow[rr, bend right] & \leq & (A \leftharpoonup B) \times (A \leftharpoonup C) \arrow[ll, bend right, "\mathsf{peither}"'] \end{tikzcd}$$

with

$$\mathsf{peither} = [\ ,\ ]$$

where $[R, S] = (R \cdot \iota_1^\circ) \cup (S \cdot \iota_2^\circ)$.

The state space factorization underlying this sort of laws may lead to a similar component factorization by aggregation of the original actions according to the part of the state space they manipulate. This finds application in typical re-engineering processes in which clusters of related operations identified in monolythic code are coupled together around specific state loci. The process is suggested in the following diagram where data refinement induces the factorization of the original component into two new ones which are composed in parallel.


$\mathsf{C}_{(B\times C)\leftharpoondown A}$
$\mathsf{C}_{B\leftharpoondown A}$
$\boxtimes$
$\mathsf{C}_{C\leftharpoondown A}$

Our main current research concern is the study of a precise characterization of this phenomonom. In particular, a suitable approach entails the need for *rethinking interfaces* in terms of decomposition of operations' signatures into pairs of input/output *ports* (as in *e.g.*, [10]) providing a basis for the specification of *component usage* as a transition structure over port names. In this context, representation *invariants* induced by data refinement (notice that both join and cojoin abstractions are partial) would generate aditional constraints over such *component usage* specifications.

Acknowledgements.

This research was funded by the Portuguese Foundation for Science and Technology, in the context of the PURe project, under contract POSI/ICHS/44304/2002. The on-going collaboration with José Oliveira on the refinement problem is gratefully acknowledged.

References

1. J. R. Abrial. *The* B *Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
2. P. Aczel and N. Mendler. A final coalgebra theorem. In D. Pitt, D. Rydeheard, P. Dybjer, A. Pitts, and A. Poigne, editors, *Proc. Category Theory and Computer Science*, pages 357–365. Springer Lect. Notes Comp. Sci. (389), 1988.
3. F. Arbab. Abstract behaviour types: a foundation model for components and their composition. In F. S. de Boer, M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Proc. First International Symposium on Formal Methods for Components and Objects (FMCO'02)*, pages 33–70. Springer Lect. Notes Comp. Sci. (2852), 2003.
4. R. C. Backhouse and P. F. Hoogendijk. Elements of a relational theory of datatypes. In B. Möller, H. Partsch, and S. Schuman, editors, *Formal Program Development*, pages 7–42. Springer Lect. Notes Comp. Sci. (755), 1993.
5. 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.

6. L. S. Barbosa. Components as processes: An exercise in coalgebraic modeling. In S. F. Smith and C. L. Talcott, editors, *FMOODS'2000 - Formal Methods for Open Object-Oriented Distributed Systems*, pages 397–417. Kluwer Academic Publishers, September 2000.
7. L. S. Barbosa. Towards a Calculus of State-based Software Components. *Journal of Universal Computer Science*, 9(8):891–909, August 2003.
8. L. S. Barbosa and J. N. Oliveira. State-based components made generic. In H. Peter Gumm, editor, *CMCS'03, Elect. Notes in Theor. Comp. Sci.*, volume 82.1. Elsevier, 2003.
9. L. S. Barbosa, M. Sun, B. K. Aichernig, and N. Rodrigues. On the semantics of componentware: a coalgebraic perspective. In Jifeng He and Zhiming Liu, editors, *Mathematical Frameworks for Component Software: Models for Analysis and Synthesis*, Series on Component-Based Development. World Scientific, 2005.
10. M. A. Barbosa and L. S. Barbosa. Specifying software connectors. In K. Araki and Z. Liu, editors, *1st International Colloquium on Theorectical Aspects of Computing (ICTAC'04)*, pages 53–68, Guiyang, China, September 2004. Springer Lect. Notes Comp. Sci. (3407).
11. J. Benabou. Introduction to bicategories. *Springer Lect. Notes Maths. (47)*, pages 1–77, 1967.
12. J. van Benthem. *Modal Correspondence Theory*. Ph.D. thesis, University of Amsterdam, 1976.
13. K. Bergner, A. Rausch, M. Sihling, A. Vilbig, and M. Broy. A Formal Model for Componentware. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 189–210. Cambridge University Press, 2000.
14. R. Bird. *Functional Programming Using Haskell*. Series in Computer Science. Prentice-Hall International, 1998.
15. R. Bird and O. Moor. *The Algebra of Programming*. Series in Computer Science. Prentice-Hall International, 1997.
16. M. Broy. Semantics of finite and infinite networks of communicating agents. *Distributed Computing*, (2), 1987.
17. D. Gelernter and N. Carrier. Coordination languages and their significance. *Communication of the ACM*, 2(35):97–107, February 1992.
18. C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.
19. P. F. Hoogendijk. *A generic theory of datatypes*. Ph.D. thesis, Department of Computing Science, Eindhoven University of Technology, 1996.
20. B. Jacobs. Objects and classes, co-algebraically. In C. Lengauer B. Freitag, C.B. Jones and H.-J. Schek, editors, *Object-Orientation with Parallelism and Persistence*, pages 83–103. Kluwer Academic Publishers, 1996.
21. B. Jacobs and J. Hughes. Simulations in coalgebra. In H. Peter Gumm, editor, *CMCS'03, Elect. Notes in Theor. Comp. Sci.*, volume 82.1, Warsaw, April 2003.
22. Bart Jacobs. Exercises in coalgebraic specification. In R. Backhouse, R. Crole, and J. Gibbons, editors, *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, pages 237–280. Springer Lect. Notes Comp. Sci. (2297), 2002.
23. Cliff B. Jones. *Systematic Software Development Using* VDM. Series in Computer Science. Prentice-Hall International, 1986.
24. P. Katis, N. Sabadini, and R. F. C. Walters. Bicategories of processes. *Journal of Pure and Applied Algebra*, 115(2):141–178, 1997.

25. 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.
26. A. Kock. Strong functors and monoidal monads. *Archiv für Mathematik*, 23:113–120, 1972.
27. G. H. Mealy. A method for synthesizing sequential circuits. *Bell Systems Techn. Jour.*, 34(5):1045–1079, 1955.
28. E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Proceedings of the 1991 ACM Conference on Functional Programming Languages and Computer Architecture*, pages 124–144. Springer Lect. Notes Comp. Sci. (523), 1991.
29. Sun Meng and L. S. Barbosa. On refinement of generic software components. In C. Rettray, S. Maharaj, and C. Shankland, editors, *10th Int. Conf. Algebraic Methods and Software Technology (AMAST)*, pages 506–520, Stirling, 2004. Springer Lect. Notes Comp. Sci. (3116). Best Student Co-authored paper Award.
30. Sun Meng and L. S. Barbosa. Components as coalgebras: The refinement dimension. *Theor. Comp. Sci. (accepted for publication)*, 2005.
31. R. Milner. *Communication and Concurrency.* Series in Computer Science. Prentice-Hall International, 1989.
32. O. Nierstrasz and F. Achermann. A calculus for modeling software components. In F. S. de Boer, M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Proc. First International Symposium on Formal Methods for Components and Objects (FMCO'02)*, pages 339–360. Springer Lect. Notes Comp. Sci. (2852), 2003.
33. S. Oaks and H. Wong. *Jini in a Nutshell.* O'Reilly and Associates, 2000.
34. J. N. Oliveira. A reification calculus for model-oriented software specification. *Formal Aspects of Computing*, 2(1):1–23, 1990.
35. J. N. Oliveira. Software reification using the SETS calculus. In *Proc. of the BCS FACS 5th Refinement Workshop, Theory and Practice of Formal Software Development, London, UK*, pages 140–171. Springer-Verlag, 8–10 January 1992. (Invited paper).
36. J. N. Oliveira and C. J. Rodrigues. Transposing relations: From *Maybe* functions to hash tables. In D. Kozen, editor, *7th International Conference on Mathematics of Program Construction*, pages 334–356. Springer Lect. Notes Comp. Sci. (3125), July 2004.
37. G. Papadopoulos and F. Arbab. Coordination models and languages. In *Advances in Computers — The Engineering of Large Systems*, volume 46, pages 329–400. 1998.
38. D. Park. Concurrency and automata on infinite sequences. pages 561–572. Springer Lect. Notes Comp. Sci. (104), 1981.
39. Erik Poll. A coalgebraic semantics of subtyping. *Theorectical Informatica and Apllications*, 35(1):61–82, 2001.
40. H. Reichel. An approach to object semantics based on terminal co-algebras. *Math. Struct. in Comp. Sci.*, 5:129–152, 1995.
41. W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*, volume 47 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1998.
42. 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).
43. J.-G. Schneider and O. Nierstrasz. Components, scripts, glue. In L. Barroca, J. Hall, and P. Hall, editors, *Software Architectures - Advances and Applications*, pages 13–25. Springer-Verlag, 1999.

44. K. Segerberg. An essay in classical modal logic. *Filosofiska Studier*, (13), 1971.
45. J. M. Spivey. *The* Z *Notation: A Reference Manual (2nd ed)*. Series in Computer Science. Prentice-Hall International, 1992.
46. C. Szyperski. *Component Software, Beyond Object-Oriented Programming*. Addison-Wesley, 1998.
47. The RAISE Language Group. *The RAISE Specification Language*. Prentice Hall International, 1992.
48. D. Turi and J. Rutten. On the foundations of final coalgebra semantics: non-well-founded sets, partial orders, metric spaces. *Math. Struct. in Comp. Sci.*, 8(5):481–540, 1998.
49. P. Wadler and K. Weihe. Component-based programming under different paradigms. Technical report, Dagstuhl Seminar 99081, February 1999.