Journal of Universal Computer Science, vol. 10, no. 7 (2004), 808-823
submitted: 16/2/04, accepted: 21/6/04, appeared: 28/7/04 © *J.UCS*

A Relational Model for Component Interconnection

Marco Antonio Barbosa
(Dep. Informática, Universidade do Minho, Portugal
marco.antonio@di.uminho.pt)

Luís Soares Barbosa
(Dep. Informática, Universidade do Minho, Portugal
lsb@di.uminho.pt)

Abstract: The basic motivation of component based development is to replace conventional programming by the composition of reusable off-the-shelf units, externally coordinated through a network of connecting devices, to achieve a common goal. This paper introduces a new *relational* model for software connectors and discusses some preliminary work on its implementation in HASKELL. The proposed model adopts a coordination point of view in order to deal with components' temporal and spatial decoupling and, therefore, to provide support for looser levels of inter-component dependency and effective external control.
Key Words: coordination models, relations, coalgebra
Category: D.3.1, F.3.2

1 Introduction

An increasing number of computer based systems are based on the cooperation of distributed, heterogeneous components, organized into open software architectures, that are expected to survive in loosely-coupled environments and be easily adapted to changing application requirements. The expression *component-based programming*, although it has been around for a long time, became a buzzword in mid 1990's (see, e.g., [Nierstrasz and Dami, 1995, Szyperski, 1998]). The basic motivation is to replace conventional programming by the composition and configuration of reusable off–the–shelf units, often regarded as *'abstractions with plugs'*. In this sense, a *component* is a 'black-box' entity which both provides and requires services, encapsulated through a public interface, which may exhibit both static and behavioural information.

There are essentially two ways of regarding, conceptually, *component-based* software development. The most wide-spread, which underlies popular technologies like, *e.g.*, CORBA, DCOM or JAVABEANS, reflects what could be called the *object orientation* legacy. A component, in this sense, is essentially a collection of objects and, therefore, component interaction is achieved by mechanisms implementing the usual *method call* semantics. As F. Arbab stresses in [Arbab, 2003] this

> induces an asymmetric, unidirectional semantic dependency of users (of services) on providers (...) which subverts independence of components, contributes to the breaking of their encapsulation, and leads to a level of inter-dependence among components that is no looser than that among objects within a component.

An alternative point of view is inspired by research on coordination languages [Gelernter and Carrier, 1992, Papadopoulos and Arbab, 1998] and favours strict component decoupling in order to support a looser inter-component dependency. In this view, computation and coordination are clearly separated, communication becomes *anonymous* and component interconnection is externally controled. This model is (partially) implemented in JavaSpaces on top of Jini and fundamental to a number of approaches to component-based development which identify communication by generic channels as the basic interaction mechanism — see, *e.g.*, Reo [Arbab, 2003] or Piccola [Nierstrasz and Achermann, 2003].

This paper adopts the latter point of view, introducing a very simple relational model for component *connectors* — the basic entities used to build appropriate *gluing code* to orchestrate components of different origins and purposes. We assume that messages are not only anonymous but also free from any sort of control information (which prevents them from being interpreted as a method invocation or an event occurrence). They just flow through the connector network in which the usual *send* or *receive* operations are, respectively, simple *read* or *write* actions on connectors' (or components') *ports*. The model, which is introduced in section 2, is close to F. Arbab's approach (as documented in, *e.g.*, [Arbab, 2003] and [Arbab and Rutten, 2002]), but for a fundamental difference. While Arbab's model specifies channels as relations between *streams*, *i.e.*, infinite sequences, of messages, we resort to simple *one-step* relations between single data elements and time tags. This leads to a simpler calculus which exploits the power of the algebra of binary relations [Backhouse and Hoogendijk, 1993], applied to an elementary data domain. That infinite behaviour, amenable to full coinductive reasoning, can be recovered from such a simpler model, is briefly discussed in section 4. Section 3 reports some preliminary work on prototyping component connectors in Haskell.

Notation. The paper resorts to a quite standard mathematical notation to express sets, functions and relations. Because *relations* are probably less familiar to the working software engineer than, say, functions, let us briefly provide a basic introduction.

Relations. Let $R : B \longleftarrow A$ denote a binary relation on (source) type A and (target) type B, and bRa stand for the representation of $\langle b, a\rangle \in R$. The set of relations from A to B is *ordered* by inclusion $\subseteq$, with relation equality being established by anti-symmetry. Fact $R \subseteq S$ means that relation S is either more defined or less deterministic than R, that is, for all a and b of the appropriate types, $bRa \Rightarrow bSa$.

The algebra of relations is built on top of three basic operators: composition ($R \cdot S$), meet ($R \cap S$) and converse (R°). As expected, $aR^\circ b$ iff bRa, meet corresponds to set-theorectical intersection and $\cdot$ generalizes functional composition: $b(R \cdot S)c$ holds iff there exists some $a \in A$ such that $bRa \wedge aSc$.
Any function f can be seen as the relation given by its graph, which, in this paper, is also denoted by f. Therefore $bfa \equiv b = fa$. In this setting functions enjoy a number of properties of which the following is singled out by its role in the pointwise to pointfree conversion:

$$b\,(f^\circ \cdot R \cdot g)\,a \;\equiv\; (fb)\,R\,(ga) \tag{1}$$

Conversely, any relation $R : B \longleftarrow A$ can be uniquely transposed into a set-valued function $\Lambda R : \mathcal{P}B \longleftarrow A$, where the transpose operator Λ satisfies the following universal property:

$$f = \Lambda R \;\equiv\; (bRa \;\equiv\; b \in (fa)) \tag{2}$$

We denote by Rel the category of sets and binary relations. References [Bird and Moor, 1997] and, mainly, [Backhouse and Hoogendijk, 1993], provide a detailed account of the calculus of binary relations, in a *pointfree* calculational style.

2 Connectors as Relations

Software components interact through anonymous messages flowing through a connector network. The basic intuition, borrowed from the coordination paradigm, is that connectors and components are independent devices, which make the latter amenable to external coordination control by the former.
Connectors have *interface points*, or *ports*, through which messages flow. Each port has an *interaction polarity* (either *input* or *output*), but, in general, connectors are blind with respect to the data values flowing through them. Consequently, let us assume $\mathbb{D}$ as the generic type of such values. The model also assumes that, on crossing the borders of a connector, every data value becomes labelled by a *time stamp* which represents a (rather weak) notion of time intended to express *order of occurrence*. As in [Arbab, 2003], temporal *simultaneity* is simply understood as *atomicity*, in the sense that two equally tagged input or output events are supposed to occur in an atomic way, that is, without being interleaved by other events.

2.1 Connectors

Let C be a connector with m input and n output ports. Its semantics is given by a relation

$$[\![C]\!] : (\mathbb{D} \times \mathbb{T})^n \longleftarrow (\mathbb{D} \times \mathbb{T})^m \tag{3}$$

where $\langle \mathbb{T}, \leq \rangle$ is a total order acting as the domain of time tags. A relation being, by definition, a set of ordered pairs, we may split $[\![C]\!]$ into two relations: one, $\mathsf{data}.[\![C]\!] : \mathbb{D}^n \longleftarrow \mathbb{D}^m$, over the data values and another, $\mathsf{time}.[\![C]\!] : \mathbb{T}^n \longleftarrow \mathbb{T}^m$, over the time tags, as follows,

$$\mathsf{data}.[\![C]\!] \;=\; \mathcal{P}(\pi_1 \cdot \mathsf{m})[\![C]\!] \qquad \text{and} \qquad \mathsf{time}.[\![C]\!] \;=\; \mathcal{P}(\pi_2 \cdot \mathsf{m})[\![C]\!] \tag{4}$$

where $\mathcal{P} f\, s$ is the map of function f on set s, π_1, π_2 are the first and second projections associated to a cartesian product and $\mathsf{m} : (\mathbb{D}^n \times \mathbb{D}^m) \times (\mathbb{T}^n \times \mathbb{T}^m) \longleftarrow (\mathbb{D} \times \mathbb{T})^n \times (\mathbb{D} \times \mathbb{T})^m$ is a parameter re-arranjement isomorphism, indexed on m, n.

Channels. The most elementary connector is the *synchronous channel* with two ports of opposite polarity. Its semantics is simply the identity relation on the time-tagged domain $\mathbb{D} \times \mathbb{T}$:

$$[\![\, \bullet \longmapsto \bullet \,]\!] \;=\; \mathsf{Id}_{\mathbb{D}\times\mathbb{T}} \tag{5}$$

which forces input and output to become mutually blocking, in the sense that any of them must wait for the other to be completed. The synchronous channel, however, is just a special case, for $f = \mathsf{id}$, of a more generic connector with the ability to perform, in a systematic way, any kind of data conversion on the flow of messages. For any function $f : \mathbb{D} \longleftarrow \mathbb{D}$, the corresponding *transformer* is defined by

$$\mathsf{data}.[\![\, \bullet \xmapsto{\ulcorner f \urcorner} \bullet \,]\!] \;=\; f \qquad \text{and} \qquad \mathsf{time}.[\![\, \bullet \xmapsto{\ulcorner f \urcorner} \bullet \,]\!] \;=\; \mathsf{Id}_{\mathbb{T}} \tag{6}$$

where f in the right hand side is the *relation* denoting the graph of *function* f. Again synchrony is forced by the specification of the time relation as the identity. If both synchrony and the accurate delivery of messages are specified by identity relations, any correflexive relation, that is any subset of the identity, provides channels which can loose information. Such channels can model, for example, unreliable communications. Therefore, we define, a *lossy channel* as

$$[\![\, \bullet \xmapsto{\cdots} \bullet \,]\!] \;\subseteq\; \mathsf{Id}_{\mathbb{D}\times\mathbb{T}} \tag{7}$$

A *filter* is an example of a lossy channel in which some messages are discarded in a controlled way, according to a given predicate $\phi : \mathbf{2} \longleftarrow \mathbb{D}$. Noting that any predicate ϕ can be seen as a relation $\mathsf{R}_\phi : \mathbb{D} \longleftarrow \mathbb{D}$ such that $d \mathsf{R}_\phi d'$ iff $d = d' \,\wedge\, (\phi\, d)$, define

$$\mathsf{data}.[\![\, \bullet \xmapsto{\ulcorner \phi \urcorner} \bullet \,]\!] \;=\; \mathsf{R}_\phi \qquad \text{and} \qquad \mathsf{time}.[\![\, \bullet \xmapsto{\ulcorner \phi \urcorner} \bullet \,]\!] \;=\; \mathsf{Id}_{\mathbb{T}} \tag{8}$$

Sources and Sinks. For each value $d \in \mathbb{D}$, a *source* $\Diamond_d$ is a device which permanentely outputs d. It has only one output port, therefore, $[\![\Diamond_d]\!] : \mathbb{D} \times \mathbb{T} \longleftarrow$

1. Clearly, the transpose of this relation gives a set $\Lambda\,[\![\Diamond_d]\!] : \mathcal{P}(\mathbb{D}\times\mathbb{T}) \longleftarrow \mathbf{1}$, defined by

$$(\mathcal{P}\pi_1)\,(\Lambda\,[\![\Diamond_d]\!]) = \{d\} \quad \text{and} \quad (\mathcal{P}\pi_2)\,(\Lambda\,[\![\Diamond_d]\!]) = \mathbb{T} \tag{9}$$

Dually, a *sink* has only an input port which accepts, and discards, any possible message. Therefore, $[\![\blacklozenge]\!] : \mathbf{1} \longleftarrow \mathbb{D}\times\mathbb{T}$, whose functional transpose, $\Lambda\,[\![\blacklozenge]\!] : \mathcal{P}\mathbf{1} \longleftarrow \mathbb{D}\times\mathbb{T}$, is simply a predicate (because $\mathcal{P}\mathbf{1} \cong \mathbf{2}$). Therefore, define

$$[\![\blacklozenge]\!] \;=\; \underline{\mathsf{true}} \tag{10}$$

Drains. A *drain* $\bullet\!\vdash\!\overset{\blacktriangledown}{\longrightarrow}\!\dashv\!\bullet \;:\; \mathbf{1} \longleftarrow (\mathbb{D}\times\mathbb{T})^2$ has two input, but no output, ports. This means that every message dropped at one of its ports is simply lost. Drains, however, can be classified according to their synchronization discipline. A drain is *synchronous* if both write operations are requested to succeed at the same time (which implies that each write attempt remains pending until another write occurs in the other end-point). It is *asynchronous* if, on the other hand, write operations in the two ports do not coincide. The formal definitions are, respectively,

$$\Lambda\,[\![\,\bullet\!\vdash\!\overset{\blacktriangledown}{\longrightarrow}\!\dashv\!\bullet\,]\!] \;=\; (\pi_2{\cdot}\pi_1 = \pi_2{\cdot}\pi_2) \qquad \text{and} \qquad \Lambda\,[\![\,\bullet\!\vdash\!\overset{\triangledown}{\longrightarrow}\!\dashv\!\bullet\,]\!] \;=\; (\pi_2{\cdot}\pi_1 \neq \pi_2{\cdot}\pi_2)$$

Broadcasters and Concentrators. The *broadcaster* connector replicates in each of its two output ports, any input received in its (unique) entry, as depicted below:

$$\begin{tikzcd}[column sep=small] & & \langle a,t\rangle \\ \langle a,t\rangle \arrow[r, mapsto, no head] & \bullet \arrow[ur, bend left] \arrow[dr, bend right] & \\ & & \langle a,t\rangle \end{tikzcd}$$

its semantics is, therefore, given by the diagonal relation $\triangle$: $(\mathbb{D}\times\mathbb{T})^2 \longleftarrow \mathbb{D}\times\mathbb{T}$ on $\mathbb{D}\times\mathbb{T}$, defined by $\langle y,z\rangle \;\triangle\; x$ iff $x = y = z$:

$$[\![\triangleleft]\!] \;= \triangle_{\mathbb{D}\times\mathbb{T}} \tag{11}$$

The dual of the *broadcaster* is the *concentrator* which accepts identical messages in both of its input ports to be delivered on output. Formally,

$$[\![\triangleright]\!] \;=\; [\![\triangleleft]\!]^{\circ} \tag{12}$$

Merger. Input and output in both broadcasters or concentrators is synchronous. In a number of real situations, however, there is a need for a connector with the ability to accept messages arriving asynchronously from different sources and merge them into an unique output. The *merger* connector is depicted as a *concentrator*

$$\begin{tikzcd}
y \arrow[dr, bend left] & & \\
& \bullet \arrow[r, mapsto] & x \\
z \arrow[ur, bend right] & &
\end{tikzcd}$$

but with a totally different semantics:

$$x \,[\![\blacktriangleright]\!]\, \langle y, z\rangle \;\equiv\; (x = y \vee x = z) \;\wedge\; \pi_2\ y \neq \pi_2\ z \tag{13}$$

Let $\phi\langle y, z\rangle$ denote the predicate $\pi_2\ y \neq \pi_2\ z$. Now note that, resorting to law (1), equation (13) can be easily converted to pointfree notation:

$$[\![\blacktriangleright]\!] \;=\; (\pi_1 \cup \pi_2) \cdot \mathsf{R}_\phi \tag{14}$$

because

$$\begin{aligned}
x \,[\![\blacktriangleright]\!]\, \langle y, z\rangle \;&\equiv\; (x = y \vee x = z) \;\wedge\; \phi(y, z) \\
&\equiv\; (x = \pi_1(y, z) \vee x = \pi_2(y, z)) \;\wedge\; \phi(y, z) \\
&\equiv\; (x(\mathsf{id}^\circ \cdot \pi_1)(y, z) \vee x(\mathsf{id}^\circ \cdot \pi_2))(y, z) \;\wedge\; \phi(y, z) \\
&\equiv\; x\,((\pi_1 \cup \pi_2) \cdot \mathsf{R}_\phi)\,\langle y, z\rangle
\end{aligned}$$

Postponer. Temporal adjustments, through the introduction of delays, are often required in coordination situations. In our model, a *postponer* is specified as

$$\mathsf{data}.[\![\, \bullet \overset{\delta}{\longmapsto} \bullet \,]\!] \;=\; \mathsf{Id}_{\mathbb{D}} \qquad \text{and} \qquad \mathsf{time}.[\![\, \bullet \overset{\delta}{\longmapsto} \bullet \,]\!] \;= > \tag{15}$$

Repeater. On reception of message $\langle a, t\rangle$ the *repeater* outputs a on all subsequent values of time, until, eventually, a new message arrives. Formally, $\mathsf{data}.[\![\, \bullet \overset{\rho}{\longmapsto} \bullet \,]\!] \;=\; \mathsf{Id}_{\mathbb{D}}$ and $\mathsf{time}.[\![\, \bullet \overset{\rho}{\longmapsto} \bullet \,]\!]$ is characterized by the following predicate

$$\psi \;=\; \forall_{t \in \mathcal{P}(\pi_1)(\mathsf{time}.[\![\, \bullet \overset{\rho}{\longmapsto} \bullet \,]\!])} \;.\; \{t' \mid \langle t', t\rangle \in \mathsf{time}.[\![\, \bullet \overset{\rho}{\longmapsto} \bullet \,]\!]\} \;\subseteq_\star\; \uparrow t$$

where $\uparrow t$ is the principal $\leq$-filter generated by t and $s_1 \subseteq_\star s_2$, for s_1, s_2 subsets of a total order, means that, once both of them have been enumerated as ascending chains, the former is a *prefix* of the latter.

2.2 New Connectors For Old

The basic way of combining two connectors is by plugging the output ports of one of them to the inputs of the other. Semantically, this amounts to relational composition:

$$[\![C_1 \,;\, C_2]\!] \;=\; [\![C_2]\!] \cdot [\![C_1]\!] \tag{16}$$

for C_1, C_2 with matching signatures. In the general case, however, this form of composition must be made partial, *i.e.*, connecting only a (specified) subset of ports. This is achieved by composing $[\![C_2]\!]$ and $[\![C_1]\!]$ using the partial relational composition operators

$$\begin{array}{ll} R \cdot_{+} S : B \longleftarrow C \times A & \text{for } \; R : B \longleftarrow C \times D, S : D \longleftarrow A \\ R \cdot^{+} S : B \times C \longleftarrow A & \text{for } \; R : B \longleftarrow D, S : C \times D \longleftarrow A \end{array}$$

with the obvious definitions[1], which, on connectors, correspond to operators $;_{+}$ and $;^{+}$, respectively.

The other aggregation scheme is *parallel* composition whose semantics is given by relational *product*, in the general case, or relational *split*, when both connectors have identical input signatures,

$$[\![C_1 \boxtimes C_2]\!] \;=\; [\![C_1]\!] \times [\![C_2]\!] \qquad \text{and} \qquad [\![\langle C_1, C_2\rangle]\!] \;=\; \langle [\![C_1]\!], [\![C_2]\!]\rangle$$

where $\times$ and $\langle\ ,\ \rangle$ are both *relators* in Rel [2].
Computing their pointwise definition may help to build up one's intuition. For example,

$$\begin{array}{rcl} \langle y,z\rangle\langle [\![C_1]\!], [\![C_2]\!]\rangle x &=& \langle y,z\rangle(\pi_1^\circ \cdot [\![C_1]\!] \,\cap\, \pi_2^\circ \cdot [\![C_2]\!])x \\ &=& \langle y,z\rangle(\pi_1^\circ \cdot [\![C_1]\!])x \;\wedge\; \langle y,z\rangle(\pi_2^\circ \cdot [\![C_2]\!])x \\ &=& \pi_1\langle y,z\rangle \,[\![C_1]\!]\, x \;\wedge\; \pi_2\langle y,z\rangle \,[\![C_2]\!] x \\ &=& y[\![C_1]\!]x \;\wedge\; z[\![C_2]\!]x \end{array}$$

Using *split* one may, for example, built a *broadcaster* out of two *synchronous channels*:

$$[\![\lhd]\!] = \langle [\![\, \bullet \longmapsto \bullet \,]\!], [\![\, \bullet \longmapsto \bullet \,]\!]\rangle \tag{17}$$

2.3 A Few Examples and Laws

Any formal model in Computer Science must provide reasonable answers to the following questions:

---

[1] $b(R \cdot_{+} S)\langle c, a\rangle \iff \exists_d \,.\, bR\langle c,d\rangle \wedge dSa$ and similarly for $R \cdot^{+} S$.

[2] For what follows it is enough to retain that a relator is just an endofunctor in Rel which, additionally, preserves inclusion $\subseteq$ and commutes with converse. Relational product and split are defined as $R \times S = \langle R \cdot \pi_1, S \cdot \pi_2\rangle$ and $\langle R, S\rangle = \pi_1^\circ \cdot R \cap \pi_2^\circ \cdot S$, respectively.

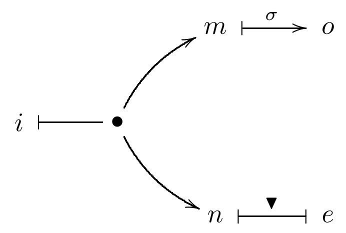
– How *expressive* is it (in our case, what kind of coordination schemes can be expressed within it)?

– How easy it is to *reason* within the model (to prove properties of such schemes)?

– How can it guide an effective *implementation* in the programming practice?

The last question is dealt in section 3. For the moment, however, and in order to illustrate the expressive power of the proposed model, consider the following example of a typical coordination pattern:

External Control Flow. The aim of this pattern is to assure that the flow of messages in a synchronous channel σ is externally controled, that is, a control signal, produced by an external source, is required for a received message to be delivered at the output end-point. Within our model this is achieved by directing messages to the input of a *broadcaster* whose output ports are connected to channel σ and to a *synchronous drain* which, on its turn, accepts the control signals. The resulting picture is



where $i, o, m, n \in \mathbb{D} \times \mathbb{T}$ stand for the messages present at the different points within the connector. Formally, the new pattern is given by the following expression in the connector algebra:

$$\lhd \;;_+\; ((\bullet \overset{\blacktriangledown}{\longmapsto\!\!\dashv} \bullet) \boxtimes (\bullet \overset{\sigma}{\longmapsto} \bullet)) \tag{18}$$

The intuition on the correctness of this scheme is that, because, both the outputs of the *broadcaster* and the two end-points of the *drain* are synchronized, the read operation on channel σ is completed simultaneously with the writing of the control signal on the free end-point of the *drain*. The reason for choosing a *drain* is simply that the actual contents of control messages is irrelevant in this context. Formally, one may argue that the semantics of **;** and of the basic connectors involved entails

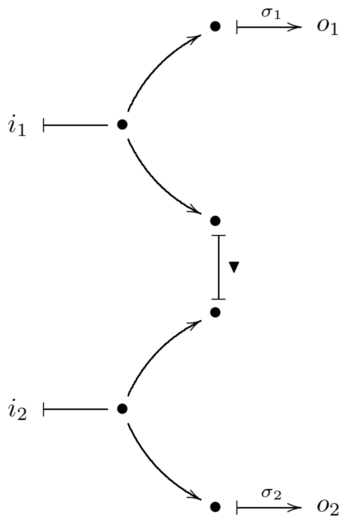
$$\begin{aligned}
(m, n)\, [\![\lhd]\!]\, i &\;\Rightarrow\; i = m = n \\
n\, [\![\, \bullet \overset{\blacktriangledown}{\longmapsto\!\!\dashv} \bullet \,]\!]\, e &\;\Rightarrow\; \pi_2\, n = \pi_2\, e \\
m\, [\![\, \bullet \overset{\sigma}{\longmapsto} \bullet \,]\!]\, o &\;\Rightarrow\; m = o
\end{aligned}$$

which leads to the desired conclusion on the time tags of o and e, that is, $\pi_2\ o = \pi_2\ e$.

Synchronization Barrier. The scheme above can be adapted to implement what is called in the coordination literature a *synchronization barrier*, that is, the enforcing of mutual synchronization between two channels. Expression

$$(\lhd \boxtimes \lhd)\ ;\ ((\bullet \overset{\sigma_1}{\longmapsto} \bullet) \boxtimes (\bullet \overset{\blacktriangledown}{\longmapsto} \bullet) \boxtimes (\bullet \overset{\sigma_2}{\longmapsto} \bullet)) \tag{19}$$

represents such a system and may be depicted as



Connector Laws. As glimpsed in the examples above, an algebra of connectors begins to emerge in which a variety of coordination patterns can be expressed. Moreover connectors in this model enjoy a number of properties generically aplicable to reason and transform such patterns. Their validity is easily established by simple computations within the relational calculus. Let us look briefly into some of them.

– First notice that a *synchronous channel* acts as the identity for connector composition,

$$(C\ ;\ \bullet \longmapsto \bullet) \quad = \quad C \quad = \quad (\bullet \longmapsto \bullet\ ;\ C) \tag{20}$$

for C with a matching signature. As **;** inherits associativity from composition in Rel, the algebra has, at least, the structure of a category.

– Similarly, a *lossy channel* acts as an absorving element for sequential composition with any kind of synchronous channels σ (including *filters*):

$$(\bullet \overset{\cdots}{\longmapsto} \bullet\ ;\ \bullet \overset{\sigma}{\longmapsto} \bullet) \quad = \quad (\bullet \overset{\sigma}{\longmapsto} \bullet\ ;\ \bullet \overset{\cdots}{\longmapsto} \bullet) \quad = \quad \bullet \overset{\cdots}{\longmapsto} \bullet \tag{21}$$

– The following laws state the expected behaviour of *transformers* and *filters* composition:

$$(\bullet \overset{\ulcorner f \urcorner}{\longmapsto} \bullet \;\; ; \;\; \bullet \overset{\ulcorner g \urcorner}{\longmapsto} \bullet) \quad = \quad \bullet \overset{\ulcorner g \cdot f \urcorner}{\longmapsto} \bullet \tag{22}$$

$$(\bullet \overset{\mathsf{R}_\phi}{\longmapsto} \bullet \;\; ; \;\; \bullet \overset{\mathsf{R}_\psi}{\longmapsto} \bullet) \quad = \quad \bullet \overset{\mathsf{R}_{\phi \wedge \psi}}{\longmapsto} \bullet \tag{23}$$

– A *synchronous channel* can be implemented by the composition of a *broadcaster* and a *concentrator*:

$$(\triangleleft \;;\; \triangleright) \quad = \quad \bullet \longmapsto \bullet \tag{24}$$

However, replacing a *concentrator* by a *merger* in the same pattern leads to *deadlock* because of incompatible synchronization policies:

$$(\triangleleft \,;\; \blacktriangleright) \quad = \quad | \tag{25}$$

where $|: \mathbf{1} \longleftarrow \mathbf{1}$ represents *deadlock* as a special connector whose semantics is simply the empty relation, *i.e.*, $[\![|]\!] = \emptyset$.

– As a last example, the following law shows how an *asynchronous drain* can be realized in two alternative ways:

$$\bullet \overset{\triangledown}{\longmapsto\!\!\!\dashv} \bullet \quad = \quad \blacktriangleright \;;\; \triangleleft \;;\; \bullet \overset{\blacktriangledown}{\longmapsto\!\!\!\dashv} \bullet \quad = \quad \blacktriangleright \;;\; \blacklozenge \tag{26}$$

Spatial Extension. A major limitation of the component connectors introduced so far is the absence of buffering capacities. A typical example of a buffered connector would be an *asynchronous channel* in which reading and writing are non mutually blocking operations.

To accommodate this kind of connectors in our repertoire requires the introduction of some form of internal memory, *i.e.*, an internal state space. Let U be the type of such memory. A buffered connector is then modelled as a relation involving not only the input and output time-tagged domains, as before, but also U, that is

$$[\![C]\!] : (\mathbb{D} \times \mathbb{T}) \times U \longleftarrow U \times (\mathbb{D} \times \mathbb{T}) \tag{27}$$

which can also be represented, by transposition to the category Set of sets and set-theorectic functions, by function

$$[\![C]\!] : \mathcal{P}((\mathbb{D} \times \mathbb{T}) \times U) \longleftarrow U \times (\mathbb{D} \times \mathbb{T}) \tag{28}$$

or, in an equivalent way,

$$[\![C]\!] : \mathcal{P}((\mathbb{D} \times \mathbb{T}) \times U)^{(\mathbb{D} \times \mathbb{T})} \longleftarrow U \tag{29}$$

that is, in the form of a *coalgebra* [Rutten, 2000] for functor $\mathsf{F}X = \mathcal{P}(X \times (\mathbb{D} \times \mathbb{T}))^{(\mathbb{D}\times\mathbb{T})}$. The coalgebraic format is adequate to single out U as the connector internal state space, not externally available. For example, a channel with a single buffering capacity is modelled as a coalgebra over $U = \mathbb{D} \times \mathbb{T}$, whereas an unbounded buffer requires $U = (\mathbb{D} \times \mathbb{T})^*$, where notation X^* stands, as usual, for finite sequences of X. In both cases a write operation updates the state variable and a read operation consumes it (or, respectively, its last element).

The use of coalgebras of this type to model buffered connectors has the main advantage of being a smooth extension of the previous relational model. Actually, any relation can be seen as a coalgebra over the singleton set, *i.e.*, $U = \mathbf{1}$. Moreover, techniques of coalgebraic analysis, namely *bisimulation*, can be uniformly used to reason about both sorts of component connectors.

3 A Haskell Implementation

This section reports some preliminary work on a HASKELL implementation of the coordination model discussed above.

3.1 Connectors in Haskell

Our starting point was Concurrent HASKELL, a very expressive extension of the language proposed in [Jones et al., 1996]. This extension provides the main features present in any typical concurrent programming language, namely, processes and a notion of atomically mutable state in order to support inter–process communication and cooperation. In particular, the library offers a primitive `forkIO` to start a fresh concurrent process, as well as the ability to create, read and write mutable variables of types `MVar` or `CVar`. `CVar` is used in the construction of *synchronous channels*, while `MVar` has a similar role in the construction of *asynchronous* ones. The basic difference between them is that `MVar` holds a buffering stream, while this is absent from `CVar`, in which case buffering is limited to a single value.

Such primitives, however, are not flexible enough to support coordination schemes within the model discussed in the previous section. Therefore we have replaced the denomination *channel* by *basic connector* and defined it by the following data type:

```
data Connector a = Connector (End (Buffer a))
                             (End (Buffer a))
                             (Time_tag)
```

where, `End` is of the type `MVar` and `Buffer` is a sequence of data items. The `Buffer` type, modelling the end-points of a basic connector, can contain either

a a single value, a unbounded or a bounded buffering capacity. This allows us to construct both, *synchronous* and *asynchronous* connectors and to implement policies that regulates the behaviour of *drains* and *lossy* connectors.
In this setting, a basic connector is a one–to–one interaction scheme which provides two end-points (known as the *source* and *sink* end-points, respectively), to external connection, and a *time_tag* which contain the creation time for the connector. It is written by inserting values into the *sink* end-point, and read by removing them from the *source* end-point. The flow of data is locally *one way*: from a component (or other connector) into the target connector or from the latter to the former.
In general, however, the library allows the construction of connectors with a potentially arbitrary number of end-points. The number and orientation of such end-points define, in each case, the connector type.

3.2 Basic Operations

Connectors are equipped with a number of basic access operations: *create*, *read*, *write* and *take*. Let us describe briefly each of them.
Create. Its signature is

```
_create :: (ConType,Filter) -> IO (Connector a)
```

where,

- `ConType` defines the type of the connector to be created,
- and `Filter` is an optional parameter which regulates the data flow through an appropriate predicate (with `*` denoting the always *true* predicate).

`ConType` is specified by a string which ranges through the following values, corresponding to the basic channels introduced in section 2: `sync`, `async`, `syncfilter`, `lossy`, `syncdrain`, `asyncdrain`, `postponer` and `antecipator`. The identifiers are self-explanatory. For instance, to create an asynchronous connector with a filter of type Integer, that is, which disposes all non integer data, we declare

```
_create (async,Int)
```

The associated action is

```
con_type == sync = do {
                        end     <- newEmptyMVar;
                        source  <- newMVar end ;
                        sink    <- newMVar end ;
                        ttag    <- getClockTime;
                        return (Connector source sink ttag)
                       }
```

Similarly, to create an asynchronous drain, write

```
_create (asyncdrain,*)
```

which performs

```
con_type == sync = do {
                       end     <- newEmptyMVar;
                       source  <- newMVar end ;
                       source  <- newMVar end ;
                       ttag    <- getClockTime;
                       return (Connector source source ttag)
                      }
```

Write. The *write* operation is given by

```
_write  :: Connector a -> a -> IO ()
```

The operation suspends the performing process (a component port or a connector end-point) until the value `a` is written to the output end.
Read. On the other hand, the *read* operation is given by

```
_read :: Connector a -> IO a
```

which suspends the performing process until a value is read from the source end-point. The value read is not removed from the connector.
Take. This is the destructive variant of read, in the sense that the value read from the source end-point is actually removed from the connector.

3.3 Composing Connectors

The library provides the three basic forms of connector composition discussed above. Due to space limitations, we restrict ourselves to sketch two small examples.
Example: a producer/consumer pattern. Two connectors `AB` and `CD` are created using the operation *_create* without any specified filter:

```
con_AB = _create(sync,*)
con_CD = _create(sync,*)
```

Now suppose there are four independent active processes: P_1, P_2, P_3 and P_4, of which P_1 and P_3 are *producers* and, dually, P_2 and P_4 are *consumers*. We then link processes P_2 and P_3 using the basic operations *_read* and *_write*. As a result, one gets a connector that sends values made available by P_1 and consumes them at P_4. The corresponding Haskell code is as follows:

```
conn = do
        value <- _read con_AB
        _write con_CD value
```

Function `conn` reads a value from the *source* connector end-point, `AB` and writes it into the *sink* end-point `CD`. The result will be a flow of the data from `A` to `D`.

Example: a barrier synchronizer. A more complex, but already introduced, example is the barrier synchronizer. In this implementation such a connector is specified by a combination of four *synchronous channels* (previously grouped on two *broadcasters*) and a *synchronous drain.*

```
con_AB = _create(sync,*)
con_CD = _create(sync,*)
con_EF = _create(syncdrain,*)
con_GH = _create(sync,*)
con_GH = _create(sync,*)
```

In the code of function `conn` below, ony the behaviour of the sychronous drain is shown:

```
conn = do
        ...
        val1 <- _take c_AB
        _write c_CD val1
        val2 <- _take c_GH
        _write c_IJ val2
        ...
```

4 Conclusions and Future Work

This paper introduced a *relational* model for connectors of software components, which adopts a coordination point of view in order to deal effectively with components' temporal and spatial decoupling and support looser levels of inter-component dependency. It was also discussed its incorporation on the HASKELL programming language.

The model is in debt to previous work of F. Arbab and his team at CWI, sharing with it a number of basic intuitions. Our fundamental departure from Arbab's work (and also from other stream based models such as, *e.g.*, [Broy, 1987, Bergner et al., 2000]) is the choice of simple data level relations to model connectors, whereas other authors resort to relations over streams. Our option leads to a clearly simpler semantics. In order to be able, however, to reason effectively about the *temporal* behaviour of connectors (and, of course, the resulting coordination patterns) we need to recover something close to the stream model from our own relations. The idea leading our current work is that connector semantics can be specified at the *gene* level, describing just the one-step behaviour, because its *temporal extension* can be computed by standard mechanisms. In fact, the behaviour of a relation $R : B \longleftarrow A$ arises, by coinduction, as the unique arrow $(\Lambda R)^\omega$ which makes the following diagram commute

$$\begin{tikzcd}
(\mathcal{P}B)^\omega \arrow[rr, "\langle \mathsf{head},\mathsf{tail}\rangle"] & & \mathcal{P}B \times (\mathcal{P}B)^\omega \\
A^\omega \arrow[u, "(\Lambda R)^\omega"] \arrow[r, "\langle \mathsf{head},\mathsf{tail}\rangle"'] & A \times A^\omega \arrow[r, "\Lambda R \times \mathsf{id}"'] & \mathcal{P}B \times A^\omega \arrow[u, "\mathsf{id} \times (\Lambda R)^\omega"']
\end{tikzcd}$$

Clearly, $(\Lambda R)^\omega$ maps a stream of inputs to a stream of sets of outputs. In short, our claim is that one may safely start with a quite elementary relational model and, then,

- introduce *buffering* capacities through the specification of a non trivial state space, which amounts to provide a sort of *spatial extension* on the connectors semantics,
- and recover the temporal behaviour of component connectors, either simple or buffered, by computing their image on the corresponding *final coalgebra*, instead of reasoning at such (more complex) level from the outset [3].

A lot remains to be done at both the *theorectical* and the *programming* levels. In particular we are working on techniques for establishing, in a straightforward way, equivalence and refinement for connectors (see [Meng and Barbosa, 2004] for some preliminary results).
A particularly important application area for this model, and corresponding calculus, is the formalisation of *software architectural patterns* [Allen and Garlan, 1997, Fiadeiro and Lopes, 1997] and the study of their laws. How easily the model scales up to accommodate *dynamically reconfigurable* connecting patterns, as in, *e.g.*, [Costa and Reggio, 1997] or [Wermelinger and Fiadeiro, 1998], remains an open challenging question.

Acknowledgements

This research was carried on in the context of the PURe Project (*Program Understanding and Re-engineering*) supported by FCT under contract `POSI/ICHS/44304/2002`.

References

[Allen and Garlan, 1997] Allen, R. and Garlan, D. (1997). A formal basis for architectural connection. *ACM TOSEM*, 6(3):213–249.
[Arbab, 2003] Arbab, F. (2003). Abstract behaviour types: a foundation model for components and their composition. In de Boer, F. S., Bonsangue, M., Graf, S., and de Roever, W.-P., editors, *Proc. First International Symposium on Formal Methods for Components and Objects (FMCO'02)*, pages 33–70. Springer Lect. Notes Comp. Sci. (2852).
[Arbab and Rutten, 2002] Arbab, F. and Rutten, J. (2002). A coinductive calculus of component connectors. CWI Tech. Rep. SEN-R0216, CWI, Amsterdam. To appear in the proceedings of WADT'02.
[Backhouse and Hoogendijk, 1993] Backhouse, R. C. and Hoogendijk, P. F. (1993). Elements of a relational theory of datatypes. In Möller, B., Partsch, H., and Schuman, S., editors, *Formal Program Development*, pages 7–42. Springer Lect. Notes Comp. Sci. (755).

[3] This builds on our own previous work on coalgebraic models for software components as documented in, *e.g.*, [Barbosa and Oliveira, 2003, Barbosa, 2003].

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

[Barbosa and Oliveira, 2003] Barbosa, L. S. and Oliveira, J. N. (2003). State-based components made generic. In Gumm, H. P., editor, *CMCS'03, Elect. Notes in Theor. Comp. Sci.*, volume 82.1.

[Bergner et al., 2000] Bergner, K., Rausch, A., Sihling, M., Vilbig, A., and Broy, M. (2000). A Formal Model for Componentware. In Leavens, G. T. and Sitaraman, M., editors, *Foundations of Component-Based Systems*, pages 189–210. Cambridge University Press.

[Bird and Moor, 1997] Bird, R. and Moor, O. (1997). *The Algebra of Programming.* Series in Computer Science. Prentice-Hall International.

[Broy, 1987] Broy, M. (1987). Semantics of finite and infinite networks of communicating agents. *Distributed Computing*, (2).

[Costa and Reggio, 1997] Costa, G. and Reggio, G. (1997). Specification of abstract dynamic data types: A temporal logic approach. *Theor. Comp. Sci.*, 173(2).

[Fiadeiro and Lopes, 1997] Fiadeiro, J. and Lopes, A. (1997). Semantics of architectural connectors. In *Proc. of TAPSOFT'97*, pages 505–519. Springer Lect. Notes Comp. Sci. (1214).

[Gelernter and Carrier, 1992] Gelernter, D. and Carrier, N. (1992). Coordination languages and their significance. *Communication of the ACM*, 2(35):97–107.

[Jones et al., 1996] Jones, S. P., Gordon, A., and Finne, S. (1996). Concurrent Haskell. In *Conference Record of POPL '96: The* 23rd *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 295–308, St. Petersburg Beach, Florida.

[Meng and Barbosa, 2004] Meng, S. and Barbosa, L. (2004). On refinement of generic software components. In *10th Int. Conf. Algebraic Methods and Software Technology (AMAST)*, Stirling. Springer Lect. Notes Comp. Sci. (to appear).

[Nierstrasz and Achermann, 2003] Nierstrasz, O. and Achermann, F. (2003). A calculus for modeling software components. In de Boer, F. S., Bonsangue, M., Graf, S., and de Roever, W.-P., editors, *Proc. First International Symposium on Formal Methods for Components and Objects (FMCO'02)*, pages 339–360. Springer Lect. Notes Comp. Sci. (2852).

[Nierstrasz and Dami, 1995] Nierstrasz, O. and Dami, L. (1995). Component-oriented software technology. In Nierstrasz, O. and Tsichritzis, D., editors, *Object-Oriented Software Composition*, pages 3–28. Prentice-Hall International.

[Papadopoulos and Arbab, 1998] Papadopoulos, G. and Arbab, F. (1998). Coordination models and languages. In *Advances in Computers — The Engineering of Large Systems*, volume 46, pages 329–400.

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

[Szyperski, 1998] Szyperski, C. (1998). *Component Software, Beyond Object-Oriented Programming*. Addison-Wesley.

[Wermelinger and Fiadeiro, 1998] Wermelinger, M. and Fiadeiro, J. (1998). Connectors for mobile programs. *IEEE Trans. on Software Eng.*, 24(5):331–341.