Specifying Software Connectors

Marco Antonio Barbosa[1]
Luís Soares Barbosa[1]

Departamento de Informática – Universidade do Minho
Campus de Gualtar
4710-057 – Braga – Portugal
{marco.antonio,lsb}@di.uminho.pt

Abstract. Orchestrating software components, often from independent suppliers, became a central concern in software construction. Actually, as relevant as components themselves, are the ways in which they can be put together to interact and cooperate in order to achieve some common goal. Such is the role of the so-called software connectors: external coordination devices which ensure the flow of data and enforce synchronization constraints within a component's network. This paper introduces a new model for software connectors, based on relations extended in time, which aims to provide support for light inter-component dependency and effective external control.

1 Introduction

The expression *software connector* was coined by software architects to represent the interaction patterns among components, the latter regarded as basic computational elements or information repositories. Their aim is to mediate the communication and coordination activities among components, acting as a sort of glueing code between them. Examples range from simple channels or pipes, to event broadcasters, synchronisation barriers or even more complex structures encoding client-server protocols or hubs between databases and applications.

Although component-based development [19, 25, 15] became accepted in industry as a new effective paradigm for Software Engineering and even considered its cornerstone for the years to come, there is still a need for precise ways to document and reason about the high-level structuring decisions which define a system's software architecture.

Conceptually, there are essentially two ways of regarding *component-based* software development. The most wide-spread, which underlies popular technologies like, *e.g.*, CORBA [24], DCOM [14] or JAVABEANS [16], 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 [3] 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 [13, 21] 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 [20] and fundamental to a number of approaches to component-ware which identify communication by generic channels as the basic interaction mechanism — see, *e.g.*, Reo [3] or Piccola [23, 18].

Adopting the latter point of view, this paper focuses on the specification of software connectors either as *relations* over a temporarily labelled data domain (representing the flow of messages) or as *relations extended in time*, *i.e.*, defined with respect to a a memory of past computations encoded as an internal state space. The latter model extends the former just as a labelled transition system extends a simple relation. Formally, we resort to coalgebraic structures [22] to model such *extended relations*, pursuing a previous line of research on applying coalgebra theory to the semantics of component-based software development (see, eg, [5, 6, 17].).

The paper is organized as follows: a semantic model for software connectors is introduced in section 2 and illustrated with the specification of one of the most basic connectors: the *asynchronous channel*. The model is further developed in section 3 which introduces a systematic way of building connectors by *aggregation* of *ports* as well as two combinators encoding, respectively, a form of *concurrent* composition and a generalization of *pipelining*. Section 4 illustrates the expressiveness of this model through the discussion of some typical examples from the literature. Finally, section 5 summarizes what has been achieved and enumerates a few research questions for the future.

Notation. The paper resorts to standard mathematical notation emphasizing a *pointfree* specification style (as in, *e.g.*, [9]) which leads to more concise descriptions and increased calculation power. The underlying mathematical universe is the category of sets and set-theoretic functions whose composition and identity are denoted by $\cdot$ and id, respectively. Notation $(\phi \rightarrow f, g)$ stands for a conditional statement: if ϕ then apply function f else g. As usual, the basic set constructs are *product* $(A \times B)$, *sum*, or disjoint union, $(A + B)$ and *function space* (B^A). We denote by $\pi_1 : A \times B \longrightarrow A$ the first projection of a product and by $\iota_1 : A \longrightarrow A + B$ the first embedding in a sum (similarly for the others). Both $\times$ and $+$ extend to functions in the usual way and, being universal constructions, a canonical arrow is defined to $A \times B$ from any set C and, symmetrically, from $A + B$ to any set C, given functions $f : C \longrightarrow A, g : C \longrightarrow B$ and $l : A \longrightarrow C, h : B \longrightarrow C$, respectively. The former is called a *split* and

denoted by $\langle f, g\rangle$, the latter an *either* and denoted by $[l, h]$, satisfying

$$k = \langle f, g\rangle \;\Leftrightarrow\; \pi_1 \cdot k = f \;\wedge\; \pi_2 \cdot k = g \tag{1}$$

$$k = [l, h] \;\Leftrightarrow\; k \cdot \iota_1 = l \;\wedge\; k \cdot \iota_2 = h \tag{2}$$

Notation B^A is used to denote *function space*, *i.e.*, the set of (total) functions from A to B. It is also characterized by an universal property: for all function $f : A \times C \longrightarrow B$, there exists a unique $\overline{f} : A \longrightarrow B^C$, called the *curry* of f, such that $f = \mathsf{ev} \cdot (\overline{f} \times C)$. Finally, we also assume the existence of a few basic sets, namely $\emptyset$, the empty set and $\mathbf{1}$, the singleton set. Note they are both 'degenerate' cases of, respectively, *sum* and *product* (obtained by applying the iterated version of those combinators to a nullary argument). Given a value v of type X, the corresponding constant function is denoted by $\underline{v} : \mathbf{1} \longrightarrow x$. Of course all set constructions are made up to isomorphism. Therefore, set $\mathbb{B} = \mathbf{1} + \mathbf{1}$ is taken as the set of boolean values true and false. Finite sequences of X are denoted by X^*. Sequences are observed, as usual, by the head (head) and tail (tail) functions, and built by singleton sequence construction (singl) and concatenation ($\frown$).

2 Connectors as Coalgebras

2.1 Connectors

According to Allen and Garlan [1] an expressive notation for software connectors should have three properties. Firstly, it should allow the specification of common patterns of architectural interaction, such as remote call, pipes, event broadcasters, and shared variables. Secondly, it should scale up to the description of complex, eventually dynamic, interactions among components. For example, in describing a client–server connection we might want to say that the server must be initialized by the client before a service request becomes enabled. Finally, it should allow for fine-grained distinctions between small variations of typical interaction patterns.

In this paper a connector is regarded as a *glueing device* between software components, ensuring the flow of data and synchronization constraints. 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 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 simplest connector one can think of — the *synchronous channel* — can be modelled just as a *function* $[\![\bullet \longmapsto \bullet]\!] : \mathbb{D} \longrightarrow \mathbb{D}$. The corresponding temporal constraint — that input and output occur simultaneously — is built-in in the very notion of a function. Such is not the case, however, of an *asynchronous channel* whose synchronization constraints entails the need for the introduction of some sort of temporal

information in the model. Therefore, we assume 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 [3], 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.

In such a setting, the semantics of a connector C, with m input and n output ports, is given by a relation

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

The *asynchronous channel*, in particular, is specified by

$$[\![\bullet \longmapsto \bullet]\!] \subseteq (\mathbb{D} \times \mathbb{T}) \times (\mathbb{D} \times \mathbb{T}) = \{((d,t),(d',t')) \mid d' = d \ \wedge \ t' > t\}$$

This simple model was proposed by the authors in [7], where its expressive power and reasoning potential is discussed. Note that with the explicit representation of a temporal dimension one is able to model non trivial synchronization restrictions. Relations, on the other hand, cater for non deterministic behaviour. For example, a *lossy* channel, *i.e.*, one that can loose information, therefore modeling unreliable communication, is specified by a correflexive relation over $\mathbb{D} \times \mathbb{T}$, *i.e.*, a subset of the identity $\mathsf{Id}_{\mathbb{D}\times\mathbb{T}}$.

On the other hand it seems difficult to express in this model the FIFO requirement usually associated to an asynchronous channel. The usual way to express such constraints, requiring a fine-grain control over the flow of data, resorts to *infinite* data structures, typically *streams*, *i.e.*, infinite sequences, of messages (as in [4, 3] or [8]). An alternative, more operational, approach, to be followed in the sequel, is the introduction of some form of internal memory in the specification of connectors. Let U be its type, which, in the asynchronous channel example, is defined as a sequence of $\mathbb{D}$ values, *i.e.*, $U = \mathbb{D}^*$, representing explicitly the buffering of incoming messages. The asynchronous channel is, then, given by the specification of two ports to which two operations over $\mathbb{D}^*$, corresponding to the reception and delivery of a $\mathbb{D}$ value, are associated. The rationale is that the operations are activated by the arrival of a data element (often referred to as a message) to the port. Formally,

$$\begin{aligned} \mathsf{receive} &: \mathbb{D}^* \times D \ \rightarrow \ \mathbb{D}^* \\ &= \frown \cdot (\mathsf{id} \times \mathsf{singl}) \\ \mathsf{deliver} &: \mathbb{D}^* \ \rightarrow \ \mathbb{D}^* \times (\mathbb{D} + \mathbf{1}) \\ &= \langle \mathsf{tl}, \mathsf{hd} \rangle \end{aligned}$$

Grouping them together leads to a specification of the channel as an elementary transition structure over $\mathbb{D}^*$, *i.e.*, a pointed *coalgebra* $\langle [] \in \mathbb{D}^*, c : \mathbb{D}^* \longrightarrow (\mathbb{D}^* \times (\mathbb{D} + \mathbf{1}))^{(\mathbb{D}+\mathbf{1})} \rangle$ where

$$\begin{aligned} \overline{c} = \mathbb{D}^* \times (\mathbb{D}+1) &\xrightarrow{\mathsf{dr}} \mathbb{D}^* \times \mathbb{D} + \mathbb{D}^* \xrightarrow{\mathsf{receive}+\mathsf{deliver}} \mathbb{D}^* + \mathbb{D}^* \times (\mathbb{D}+1) \\ &\xrightarrow{\simeq} \mathbb{D}^* \times 1 + \mathbb{D}^* \times (\mathbb{D}+1) \xrightarrow{[\mathsf{id}\times\iota_2, \mathsf{id}]} \mathbb{D}^* \times (\mathbb{D}+1) \end{aligned}$$

Note how this specification meets all the exogenous synchronization constraints, including the enforcing of a strict FIFO discipline. The temporal dimension, however, is no more explicit, but *built-in* in coalgebra dynamics. We shall come back to this in section 5. For the moment, however, let us elaborate on this example to introduce a general model of software connectors as coalgebras.

2.2 The General Model

A software connector is specified by an interface which aggregates a number of *ports* represented by operations which regulate its behaviour. Each operation encodes the port reaction to a data item crossing the connector's boundary. Let U be the type of the connector's internal state space and $\mathbb{D}$ a generic data domain for messages, as before. In such a setting we single out three kinds of ports with the following signatures:

$$\mathsf{post} \ : \ U \longrightarrow U^{\mathbb{D}} \tag{4}$$

$$\mathsf{read} \ : \ U \longrightarrow (\mathbb{D} + 1) \tag{5}$$

$$\mathsf{get} \ : \ U \longrightarrow U \times (\mathbb{D} + 1) \tag{6}$$

where

- post is an input operation analogous to a write operation in conventional programming languages (see *e.g.*, [2, 21, 3]). Typically, a post port accepts data items and store them internally, in some form.
- read is a non-destructive output operation. This means that through a read port the environment might 'observe' a data item, but the connector's state space remains unchanged. Of course read is a partial operation, because there cannot be any guarantee that data is available for reading.
- get is a destructive variation of the read port. In this case the data item is not only made externally available, but also deleted from the connector's memory.

As mentioned above, connectors are formed by the aggregation of a number of post, read and get ports. According to their number and types one specific connectors with well-defined behaviours may be defined. Let us consider some possibilities.

Sources and Sinks. The most elementary connectors are those with a unique port. According to its orientation they can be classified as

- Data *sources*, specified by a single read operation

$$\Diamond_d \ = \ \langle d \in \mathbb{D}, \iota_1 : \mathbb{D} \rightarrow \mathbb{D} + \mathbf{1} \rangle \tag{7}$$

  defined over state space $U = \mathbb{D}$ and initialized with value d.

– Data *sinks*, ie, connectors which are always willing to accept any data item, discarding it immediately. The state space of data sinks is irrelevant and, therefore, modeled by the singleton set $\mathbf{1} = \{*\}$. Formally,

$$\blacklozenge \;=\; \langle * \in \mathbf{1}, ! : \mathbf{1} \to \mathbf{1}^{\mathbb{D}} \rangle \tag{8}$$

where ! is the (universal) map from any object to the (final) set $\mathbf{1}$.

Binary Connectors. Binary connectors are built by the aggregation of two ports, assuming the corresponding operations are defined over the same state space. This, in particular, enforces mutual execution of state updates.

– Consider, first, the aggregation of two read ports, denoted by read_1 and read_2, with possibly different specifications. Both of them are (non destructive) observers and, therefore, can be simultaneously offered to the environment. The result is a coalgebra simply formed by their *split*:

$$c \;=\; \langle u \in U, \langle \mathsf{read}_1, \mathsf{read}_2 \rangle : U \to (\mathbb{D}+1) \times (\mathbb{D}+1) \rangle \tag{9}$$

– On the other hand, aggregating a post to a read port results in

$$c \;=\; \langle u \in U, \langle \mathsf{post}, \mathsf{read} \rangle : U \to U^{\mathbb{D}} \times (\mathbb{D}+1) \rangle \tag{10}$$

– Replacing the read port above by a get one requires an additive aggregation to avoid the possibility of simultaneous updates leading to

$$c \;=\; \langle u \in U, \gamma_c : U \to (U \times (\mathbb{D}+1))^{\mathbb{D}+\mathbf{1}} \rangle \tag{11}$$

where[1]

$$\begin{aligned} \overline{\gamma_c} \;=\; U \times (\mathbb{D}+1) &\xrightarrow{\;\mathsf{dr}\;} U \times \mathbb{D} + U \xrightarrow{\;\overline{\mathsf{post}}+\mathsf{get}\;} U + U \times (\mathbb{D}+1) \\ &\xrightarrow{\;\simeq\;} U \times 1 + U \times (\mathbb{D}+1) \xrightarrow{\;[\mathsf{id}\times\iota_2, \mathsf{id}]\;} U \times (\mathbb{D}+1) \end{aligned}$$

Channels of different kinds are connectors of this type. Recall the asynchronous channel example above: ports identified by $\mathsf{receive}$ and $\mathsf{deliver}$ have the same signature of a post and a get, respectively. An useful variant is the *filter* connector which discards some messages according to a given predicate $\phi : \mathbf{2} \longleftarrow \mathbb{D}$. The get port is given as before, *i.e.*, $\langle \mathsf{tl}, \mathsf{hd} \rangle$, but post becomes conditional on predicate ϕ, *i.e.*,

$$\mathsf{post} \;=\; \phi \to \frown \cdot(\mathsf{id} \times \mathsf{singl}), \mathsf{id}$$

– A similar scheme is adopted for the combination of two post ports:

$$c \;=\; \langle u \in U, \gamma_c : U \to U^{\mathbb{D}+\mathbb{D}} \rangle \tag{12}$$

where

$$\begin{aligned} \overline{\gamma_c} \;= U \times (\mathbb{D}+\mathbb{D}) &\xrightarrow{\;\mathsf{dr}\;} U \times \mathbb{D} + U \times \mathbb{D} \\ &\xrightarrow{\;\overline{\mathsf{post}_1}+\overline{\mathsf{post}_2}\;} U + U \xrightarrow{\;\nabla\;} U \end{aligned}$$

[1] In the sequel dr is the right distributivity isomorphism and ∇ the codiagonal function defined as the *either* of two identities, *i.e.*, $\nabla = [\mathsf{id}, \mathsf{id}]$.

The General Case. Examples above lead to the specification of the following shape for a connector built by aggregation of P post, G get and R read ports:

$$c \;=\; \langle u \in U, \langle \gamma_c, \rho_c \rangle : U \longrightarrow (U \times (\mathbb{D}+1))^{P \times \mathbb{D} + G} \times (\mathbb{D}+1)^R \rangle \qquad (13)$$

where ρ_c is the split the R read ports, *i.e.*,

$$\rho_c : \; U \longrightarrow (\mathbb{D}+1) \times (\mathbb{D}+1) \times \ldots \times (\mathbb{D}+1) \qquad (14)$$

and, γ_c collects ports of type post or get, which are characterized by the need to perform state updates, in the uniform scheme explained above for the binary case. Note that this expression can be rewritten as

$$U \;=\; (\sum_{i \in P} U^{\mathbb{D}} + \sum_{j \in G} U \times (\mathbb{D}+1)) \times \prod_{k \in R} (\mathbb{D}+1) \qquad (15)$$

which is, however, less amenable to symbolic manipulation in proofs.

3 Combinators

In the previous section, a general model of software connectors as pointed coalgebras was introduced and their construction by port aggregation discussed. To obtain descriptions of more complex interaction patterns, however, some forms of connector composition are needed. Such is the topic of the present section in which two combinators are defined: a form of *concurrent composition* and a generalisation of *pipelining* capturing arbitrary composition of post with either read or get ports.

3.1 Concurrent Composition

Consider connectors c_1 and c_2 defined as

$$c_i = \langle u_i \in U_i, \langle \gamma_i, \rho_i \rangle : (U_i \times (\mathbb{D}+1))^{P_i \times \mathbb{D} + G_i} \times (\mathbb{D}+1)^{R_i} \rangle$$

with P_i ports of type post, R_i of type read and G_i of type get, for $i = 1, 2$. Their concurrent composition, denoted by $c_1 \boxtimes c_2$ makes externally available all c_1 and c_2 *single* primitive ports, plus *composite* ports corresponding to the simultaneous activation of post (respectively, get) ports in the two operands. Therefore, $P' = P_1 + P_2 + P_1 \times P_2$, $G' = G_1 + G_2 + G_1 \times G_2$ and $R' = R_1 + R_2$ become available in $c_1 \boxtimes c_2$ as its interface sets. Formally, define

$$c_1 \boxtimes c_2 : U' \longrightarrow (U' \times (\mathbb{D}+1))^{P' \times \mathbb{D} + G'} \times (\mathbb{D}+1)^{R'} \qquad (16)$$

where

$$\begin{aligned}
\overline{\gamma}_{c_1 \boxtimes c_2} \;=\; & U_1 \times U_2 \times (P_1 + P_2 + P_1 \times P_2) \times \mathbb{D} + (G_1 + G_2 + G_1 \times G_2) \xrightarrow{\simeq} \\
& (U_1 \times (P_1 \times \mathbb{D} + G_1) \times U_2 + U_1 \times U_2 \times (P_2 \times \mathbb{D} + G_2) + U_1 \times (P_1 \times \mathbb{D} + G_1) \times U_2 \times (P_2 \times \mathbb{D} + G_2) \\
& \xrightarrow{\gamma_1 \times \mathsf{id} + \mathsf{id} \times \gamma_2 + \gamma_1 \times \gamma_2} (U_1 \times (\mathbb{D}+1)) \times U_2 + U_1 \times (U_2 \times (\mathbb{D}+1)) + (U_1 \times (\mathbb{D}+1)) \times (U_2 \times (\mathbb{D}+1)) \\
& \xrightarrow{\simeq} U_1 \times U_2 \times (\mathbb{D}+1) + U_1 \times U_2 \times (\mathbb{D}+1) + U_1 \times U_2 \times (\mathbb{D}+1)^2 \xrightarrow{\nabla + \mathsf{id}} \\
& U_1 \times U_2 \times (\mathbb{D}+1) + U_1 \times U_2 \times (\mathbb{D}+1) \times U_2 (\mathbb{D}+1) \xrightarrow{\simeq} U_1 \times U_2 \times ((\mathbb{D}+1) + (\mathbb{D}+1))^2
\end{aligned}$$

and

$$\rho_{c_1 \boxtimes c_2} = U_1 \times U_2 \xrightarrow{\rho_1 \times \rho_2} (\mathbb{D}+1)^{R_1} \times (\mathbb{D}+1)^{R_2} \xrightarrow{\simeq} (\mathbb{D}+1)^{R_1+R_2}$$

3.2 Hook

The *hook* combinator plugs ports with opposite polarity within an arbitrary connector

$$c = \langle u \in U, \langle \gamma_c, \rho_c \rangle : U \longrightarrow (U \times (\mathbb{D}+1))^{P \times \mathbb{D} + G} \times (\mathbb{D}+1)^R \rangle$$

There are two possible plugging situations:

1. Plugging a post port p_i to a read r_j one, resulting in

$$\begin{aligned}
\rho_{c \curvearrowleft_{r_j}^{p_i}} &= \langle r_1, \ldots, r_{j-1}, r_{j+1}, \ldots, r_R \rangle \\
\overline{\gamma}_{c \curvearrowleft_{r_j}^{p_i}} &= U \times ((P-1) \times \mathbb{D} + G) \xrightarrow{\theta \times \mathsf{id}} U \times ((P-1) \times \mathbb{D} + G) \\
&\xrightarrow{\simeq} \textstyle\sum_{P-1} U \times \mathbb{D} + \sum_G U \xrightarrow{[p_1,\ldots,p_{i-1},p_{i+1},\ldots,p_p]+[g_1,\ldots,g_G]} \\
&U + U \times (\mathbb{D}+1) \xrightarrow{\simeq} U \times 1 + U \times (\mathbb{D}+1) \\
&\xrightarrow{[\mathsf{id} \times \iota_2, \mathsf{id}]} U \times (\mathbb{D}+1)
\end{aligned}$$

   where $\theta : U \to U$

$$\begin{aligned}
\theta = U &\xrightarrow{\triangle} U \times U \xrightarrow{\mathsf{id} \times r_j} U \times \mathbb{D} + 1 \\
&\xrightarrow{\simeq} U \times \mathbb{D} + U \xrightarrow{\overline{p_i} + \mathsf{id}} U + U \xrightarrow{\nabla} U
\end{aligned}$$

2. Plugging a post port p_i to a get g_j one, resulting in

$$\begin{aligned}
\rho_{c \curvearrowleft_{r_j}^{p_i}} &= \rho_c \\
\overline{\gamma}_{c \curvearrowleft_{g_j}^{p_i}} &= U \times ((P-1) \times \mathbb{D} + (G-1)) \xrightarrow{\theta \times \mathsf{id}} \\
&U \times ((P-1) \times \mathbb{D} + (G-1)) \\
&\xrightarrow{\simeq} \textstyle\sum_{P-1} U \times \mathbb{D} + \sum_{G-1} U \\
&\xrightarrow{[p_1,\ldots,p_{i-1},p_{i+1},\ldots,p_p]+[g_1,\ldots,g_{j-1},g_{j+1},\ldots,g_G]} \\
&U + U \times (\mathbb{D}+1) \xrightarrow{\simeq} U \times 1 + U \times (\mathbb{D}+1) \\
&\xrightarrow{[\mathsf{id} \times \iota_2, \mathsf{id}]} U \times (\mathbb{D}+1)
\end{aligned}$$

   where $\theta : U \to U$

$$\begin{aligned}
\theta = U &\xrightarrow{g_j} U \times (\mathbb{D}+1) \xrightarrow{\simeq} U \times \mathbb{D} + U \\
&\xrightarrow{\overline{p_i} + \mathsf{id}} U + U \xrightarrow{\nabla} U
\end{aligned}$$

Note that, according to the definition above, if the result of a reaction at a port of type read or get is of type $\mathbf{1}$, which encodes the absence of any data item to be read, the associated post is not activated and, consequently, the interaction does not become effective.
Such unsuccessful read attempt can alternatively be understood as a *pending read request*. In this case the intended semantics for interaction with the associated post port is as follows: successive read attempts are performed until communication occurs. This version of *hook* is denoted by $\lceil^{p}_{r}\ c$ and easily obtained by replacing, in the definition of θ above, step

$$U \times \mathbb{D} + U \xrightarrow{\overline{p_i}+\mathsf{id}} U + U$$

by

$$U \times \mathbb{D} + U \xrightarrow{\overline{p_i}+\theta} U + U$$

Both forms of the *hook* combinator can be applied to a whole sequence of pairs of opposite polarity ports, the definitions above extending as expected. The two combinators introduced in this section can also be put together to define a form of *sequential composition* in situations where all the post ports of the second operand (grouped in *in*) are connected to all the read and get ports of the first (grouped in *out*). It is assumed that hooks between two single ports extend smoothly to any product of ports (as arising from concurrent composition) in which they participate. Formally, we define by abbreviation

$$c_1 \mathbin{;} c_2 \stackrel{\text{abv}}{=} (c_1 \boxtimes c_2)\ \urcorner^{in}_{out} \tag{17}$$

and

$$c_1 \bowtie c_2 \stackrel{\text{abv}}{=} \lceil^{in}_{out}\ (c_1 \boxtimes c_2) \tag{18}$$

4 Examples

This section discusses how some typical software connectors can be defined in the model proposed in this paper.

4.1 Broadcasters and Mergers

Our first example is the *broadcaster*, a connector which replicates in each of its two (output) ports, any input received in its (unique) entry as depicted bellow. There are two variants of this connector denoted, respectively, by $\blacktriangleleft$ and $\triangleleft$. The first one corresponds to a *synchronous* broadcast, in the sense that the two get ports are activated simultaneously. The other one is *asynchronous*, in the sense that both of its get ports can be activated independently. The definition of $\triangleleft$ is rather straightforward as a coalgebra over $U = \mathbb{D} + \mathbf{1}$ and operations

$$\begin{aligned} \overline{\mathsf{post}} &: U \times \mathbb{D} \rightarrow U = \iota_1 \cdot \pi_2 \\ \mathsf{get}_1, \mathsf{get}_2 &: U \rightarrow U \times (\mathbb{D} + 1) = \triangle \end{aligned}$$

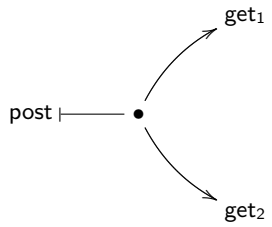

Fig. 1. The *broadcaster* connector.

where $\triangle$ is the *diagonal* function, defined by $\triangle = \langle \mathsf{id}, \mathsf{id} \rangle$. The synchronous case, however, requires the introduction of two boolean flags initialized to $\langle \mathsf{false}, \mathsf{false} \rangle$ to witness the presence of get requests at both ports. The idea is that a value is made present at both the get ports if it has been previously received, as before, and there exists two reading requests pending. Formally, let $U = (\mathbb{D} + \mathbf{1}) \times (\mathbb{B} \times \mathbb{B})$ and define

$$\begin{aligned} \overline{\mathsf{post}} \;&:\; U \times \mathbb{D} \;\to\; U \;=\; \langle \iota_1 \cdot \pi_2, \pi_2 \cdot \pi_1 \rangle \\ \mathsf{get}_1 \;&:\; U \;\to\; U \times (\mathbb{D} + 1) \;=\; (=_* \cdot \pi_1 \;\to\; \langle \mathsf{id}, \pi_1 \rangle,\, \mathsf{getaux}_1) \end{aligned}$$

where

$$\mathsf{getaux}_1 \;=\; (\pi_2 \cdot \pi_2 \;\to\; \langle (\iota_2 \cdot \underline{*}) \times (\underline{\mathsf{false}} \times \underline{\mathsf{false}}), \pi_1 \rangle,\, \langle \mathsf{id} \times (\underline{\mathsf{true}} \times \mathsf{id}), \iota_2 \cdot \underline{*} \rangle)$$

I.e., if there is no information stored flag $*$ is returned and the state left unchanged. Otherwise, an output is performed but only if there is a previous request at the other port. If this is not the case the reading request is recorded at the connector's state. This definition precludes the possibility of a reading unless there are reading requests at both ports. The fact that both requests are served depends on their interaction with the associated post ports, *i.e.*, on the chosen hook discipline (see the synchronization barrier example in subsection 4.3). The definition of get_2 is similar but for the boolean flags update:

$$\mathsf{getaux}_2 \;=\; (\pi_1 \cdot \pi_2 \;\to\; \langle (\iota_2 \cdot \underline{*}) \times (\underline{\mathsf{false}} \times \underline{\mathsf{false}}), \pi_1 \rangle,\, \langle \mathsf{id} \times (\mathsf{id} \times \underline{\mathsf{true}}), \iota_2 \cdot \underline{*} \rangle)$$

Dual to the *broadcaster* connector is the *merger* which concentrates messages arriving at any of its two post ports. The *merger*, denoted by $\triangleright$, is similar to an asynchronous channel, as given in section 2, with two identical post ports. Another variant, denoted by $\blacktriangleright$, accepts one data item a time, after which disables both post ports until get is activated. This connector is defined as a coalgebra over $U = \mathbb{D} + \mathbf{1}$ with

$$\begin{aligned} \overline{\mathsf{post}_1} = \overline{\mathsf{post}_2} \;&:\; U \times \mathbb{D} \;\to\; U \\ &=\; (=_* \cdot \pi_1 \;\to\; \pi_1,\, \iota_1 \cdot \pi_2) \\ \mathsf{get} \;&:\; U \;\to\; U \times (\mathbb{D} + \mathbf{1}) \\ &=\; (=_* \to\; \langle \triangle, \mathsf{id} \rangle,\, \langle \iota_2 \cdot \underline{*}, \mathsf{id} \rangle) \end{aligned}$$

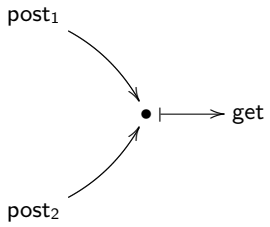

Fig. 2. The *merger* connector.

4.2 Drains

A *drain* is a symmetric connector with two inputs, but no output, points. Operationally, every message arriving to an end–point of a drain is simply lost. A drain is *synchronous* when both post operations are required to be active at the same time, and *asynchronous* otherwise. In both case, no information is saved and, therefore $U = \mathbf{1}$. Actually, drains are used to enforce synchronisation in the flow of data. Formally, an *asynchronous* drain is given by coalgebra

$$[\![\bullet \overset{\triangledown}{\longmapsto} \bullet]\!] \;:\; \mathbf{1} \longrightarrow \mathbf{1}^{\mathbb{D}+\mathbb{D}}$$

where both post ports are modelled by the (universal) function to $\mathbf{1}$, *i.e.*, $\mathsf{post}_1 = !_{U\times\mathbb{D}} = \mathsf{post}_2$. The same operations can be composed in a product to model the *synchronous* variant:

$$[\![\bullet \overset{\blacktriangledown}{\longmapsto} \bullet]\!] \;:\; U \longrightarrow U^{\mathbb{D}\times\mathbb{D}}$$

defined by

$$\mathbf{1} \times (\mathbb{D} \times \mathbb{D}) \xrightarrow{\;\cong\;} \mathbf{1} \times \mathbb{D} \times \mathbf{1} \times \mathbb{D} \xrightarrow{\overline{\mathrm{post}}_1 \times \overline{\mathrm{post}}_2} \mathbf{1} \times \mathbf{1} \xrightarrow{\;!\;} \mathbf{1}$$

There is an important point to make here. Note that in this definition two post ports were aggregated by a product, instead of resorting to the more common additive context. Such is required to enforce their simultaneous activation and, therefore, to meet the expected synchrony constraint. This type of port aggregation also appears as a result of concurrent composition. In general, when presenting a connector's interface, we shall draw a distinction between *single* and *composite* ports, the latter corresponding to the simultaneous activation of two or more of the former.

Composite ports, on the other hand, entail the need for a slight generalisation of hook. In particular it should cater for the possibility of a post port requiring, say, two values of type $\mathbb{D}$ be plugged to two (different) read or get ports. Such a generalisation is straightforward and omitted here (but used below on examples involving *drains*).

4.3 Synchronization Barrier

In the coordination literature a *synchronization barrier* is a connector used to enforce mutual synchronization between two channels (as σ_1 and σ_2 below). This is achieved by the composition of two synchronous broadcasters with two of their get ports connected by a synchronous drain. As expected, data items read at extremities o_1 and o_2 are read simultaneously. The composition pattern is depicted in figure 3, which corresponds to the following expression:

$$(\blacktriangleleft \boxtimes \blacktriangleleft) \bowtie ((\bullet \overset{\sigma_1}{\longmapsto} \bullet) \boxtimes (\bullet \longmapsto\!\blacktriangledown\, \bullet) \boxtimes (\bullet \overset{\sigma_2}{\longmapsto} \bullet)) \tag{19}$$

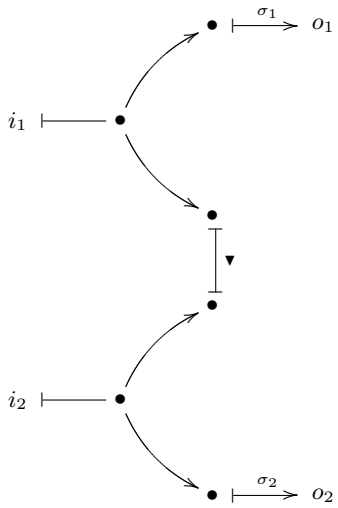

Fig. 3. A *synchronization barrier*.

4.4 The Dining Philosophers

Originally posed and solved by Dijkstra in 1965, the *dinning philosophers* problem provides a good example to experiment an exogenous coordination model of the kind proposed in this paper [2]. In the sequel we discuss two possible solutions to this problem.

[2] The basic version reads as follows. Five philosophers are seated around a table. Each philosopher has a plate of spaghetti and needs two forks to eat it. When a philosopher gets hungry, he tries to acquire his left and right fork, one at a time, in either order. If successful in acquiring two forks, he eats for a while, then puts down the forks and continues to think.

A* merger-drain *solution. One possible solution assumes the existence of five replicas of a component Phi(losopher), each one with four get ports, two on the lefthand side and another two on the righthand side. The port labeled left_i is activated by Phi_i to place a request for the left fork. On the other hand, port leftf_i is activated on its release (and similarly for the ports on the right). Coordination between them is achieved by a software connector Fork with four post ports, to be detailed below. The connection between two adjacent philosophers through a Fork is depicted below which corresponds to the following expression in the calculus

$$(Phi_i \boxtimes \mathsf{Fork}_i \boxtimes Phi_{i+1}) \; \curvearrowleft^{\mathsf{rr}_i\, \mathsf{rf}_i\, \mathsf{lr}_i\, \mathsf{lf}_i}_{\mathsf{right}_i\, \mathsf{rightf}_i\, \mathsf{left}_{i+1}\, \mathsf{leftf}_{i+1}} \tag{20}$$

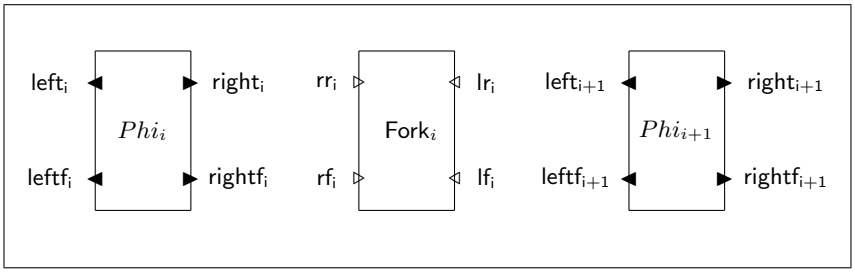

Fig. 4. Dining Philosophers (1).

The synchronization constraints of the problem are dealt by connector Fork built from two blocking mergers and a synchronous drain depicted in figure 5 and given by expression

$$(\blacktriangleright \boxplus \blacktriangleright)\,;\; \bullet \vdash\!\!\blacktriangledown\!\!\dashv \bullet \tag{21}$$

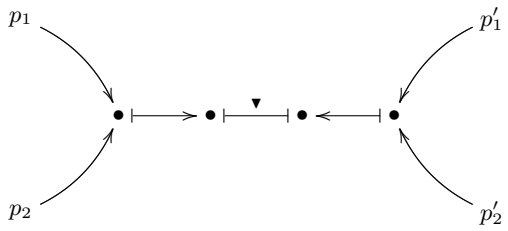

Fig. 5. A *Fork* connector (1).

A* token *solution. Another solution is based on a specification of Fork as an *exchange token* connector. Such a connector is given as a coalgebra over $U = \{\pitchfork\} + \mathbf{1}$, where $\pitchfork$ is a token representing the (physical) fork. From the point of view of a philosopher requesting a fork equivales to an attempt to remove $\pitchfork$ from the connector state space. Dually, a fork is released by returning it to the connector state space. In detail, a fork request at a philosopher port, say right, which is a post port hooked to (the get port) rr of the connector is only successful if the token is available. Otherwise the philosopher must wait until a fork is released. The port specifications for Fork are as follows

$$\begin{aligned}\overline{\mathsf{rr}} = \overline{\mathsf{lr}} \;:\; U \;&\rightarrow\; U \times (\mathbb{D} + 1)\\ &=\; (=_{\pitchfork} \rightarrow\; (\iota_2 \cdot \underline{*}) \times (\iota_1 \cdot \underline{\pitchfork}),\, \mathsf{id} \times (\iota_2 \cdot \underline{*}))\\ \overline{\mathsf{rf}} = \overline{\mathsf{lf}} \;:\; U \times \mathbb{D} \;&\rightarrow\; U\\ &=\; \iota_1 \cdot \underline{\pitchfork}\end{aligned}$$

Again, the *Fork* connector is used as a mediating agent between any two philosophers as depict in figure 6. The corresponding expression is

$$(Phi_i \boxtimes \mathsf{Fork}_i \boxtimes Phi_{i+1}) \;\leftarrow^{\mathsf{right}_i\ \mathsf{rf}_i\ \mathsf{left}_i\ \mathsf{lf}_i}_{\mathsf{rr}_i\ \mathsf{rightf}_i\ \mathsf{lr}_{i+1}\ \mathsf{leftf}_{i+1}} \tag{22}$$

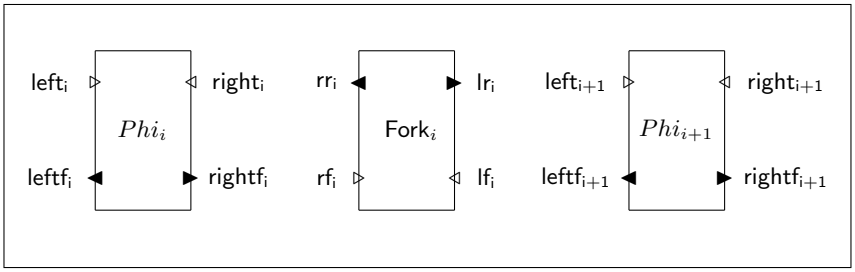

Fig. 6. Dining Philosophers (2).

5 Conclusions and Future Work

This paper discussed the formalization of software connectors, adopting a coordination oriented approach to support looser levels of inter-component dependency. Two alternative models were mentioned: relations on time-tagged domains (detailed in [7]) and (polynomial) coalgebras, regarded as relations extended in time, which is the basic issue of this paper. The close relation between the two models is still object of on-going work. In particular, how does the relational model lifts to a coalgebraic one when more complex notions of time are adopted? Note that, in most cases, the usual set-theoretic universe underlying coalgebras as used here

lacks enough structure to extend such relations over (richly structured) timed universes.

Resorting to coalgebras to specify software 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 used to reason about connectors and, in general, architectural design descriptions. In fact, although in this paper the emphasis was placed on connector modeling and expressiveness, the model supports a basic calculus in which connector equivalence and refinement can be discussed (along the lines of [17]). The model compares quite well to the more classic stream-based approaches (see *e.g.*, [10, 8, 3]), which can be recovered as the *final* interpretation of the coalgebraic specifications proposed here.

A lot of work remains to be done. Our current concerns include, in particular, the full development of a calculus of software connectors emerging from the coalgebraic semantic framework and its use in reasoning about typical *software architectural patterns* [1, 12] and their laws. How easily this work scales up to accommodate *dynamically re-configurable* architectures, as in, *e.g.*, [11] or [26], remains an open challenging question. We are also currently working on the development of an HASKELL based platform for prototyping this model, allowing the user to define and compose, in an interactive way, his/her own software connectors.

Acknowledgements. This research was carried on in the context of the PURE Project supported by FCT, the Portuguese Foundation for Science and Technology, under contract `POSI/ICHS/44304/2002`.

References

1. R. Allen and D. Garlan, *A formal basis for architectural connection*, ACM TOSEM **6** (1997), no. 3, 213–249.
2. F. Arbab, *Reo: A channel-based coordination model for component composition*, Mathematical Structures in Computer Science, **14** (2004), no. 3, 329–366.
3. ———, *Abstract behaviour types: a foundation model for components and their composition*, Proc. First International Symposium on Formal Methods for Components and Objects (FMCO'02) (F. S. de Boer, M. Bonsangue, S. Graf, and W.-P. de Roever, eds.), Springer Lect. Notes Comp. Sci. (2852), 2003, pp. 33–70.
4. F. Arbab and J. Rutten, *A coinductive calculus of component connectors*, CWI Tech. Rep. SEN-R0216, CWI, Amsterdam, 2002, To appear in the proceedings of WADT'02.
5. L. S. Barbosa, *Components as processes: An exercise in coalgebraic modeling*, FMOODS'2000 - Formal Methods for Open Object-Oriented Distributed Systems (S. F. Smith and C. L. Talcott, eds.), Kluwer Academic Publishers, September 2000, pp. 397–417.
6. ———, *Towards a Calculus of State-based Software Components*, Journal of Universal Computer Science **9** (2003), no. 8, 891–909.

7. M.A. Barbosa and L.S. Barbosa, *A Relational Model for Component Interconnection*, Journal of Universal Computer Science **10** (2004), no. 7, 808–823.
8. K. Bergner, A. Rausch, M. Sihling, A. Vilbig, and M. Broy, *A Formal Model for Componentware*, Foundations of Component-Based Systems (Gary T. Leavens and Murali Sitaraman, eds.), Cambridge University Press, 2000, pp. 189–210.
9. R. Bird and O. Moor, *The algebra of programming*, Series in Computer Science, Prentice-Hall International, 1997.
10. M. Broy, *Semantics of finite and infinite networks of communicating agents*, Distributed Computing (1987), no. 2.
11. G. Costa and G. Reggio, *Specification of abstract dynamic data types: A temporal logic approach*, Theor. Comp. Sci. **173** (1997), no. 2.
12. J. Fiadeiro and A. Lopes, *Semantics of architectural connectors*, Proc. of TAPSOFT'97, Springer Lect. Notes Comp. Sci. (1214), 1997, pp. 505–519.
13. D. Gelernter and N. Carrier, *Coordination languages and their significance*, Communication of the ACM **2** (1992), no. 35, 97–107.
14. R. Grimes, *Profissional dcom programming*, Wrox Press, 1997.
15. He Jifeng, Liu Zhiming, and Li Xiaoshan, *A contract-oriented approach to component-based programming*, Proc. of FACS'03, (*Formal Approaches to Component Software*) (Pisa) (Z. Liu, ed.), Spetember 2003.
16. V. Matena and B Stearns, *Applying entreprise javabeans: Component-based development for the j2ee platform*, Addison-Wesley, 2000.
17. Sun Meng and L. S. Barbosa, *On refinement of generic software components*, 10th Int. Conf. Algebraic Methods and Software Technology (AMAST) (Stirling) (C. Rettray, S. Maharaj, and C. Shankland, eds.), Springer Lect. Notes Comp. Sci. (3116), 2004, pp. 506–520.
18. O. Nierstrasz and F. Achermann, *A calculus for modeling software components*, Proc. First International Symposium on Formal Methods for Components and Objects (FMCO'02) (F. S. de Boer, M. Bonsangue, S. Graf, and W.-P. de Roever, eds.), Springer Lect. Notes Comp. Sci. (2852), 2003, pp. 339–360.
19. O. Nierstrasz and L. Dami, *Component-oriented software technology*, Object-Oriented Software Composition (O. Nierstrasz and D. Tsichritzis, eds.), Prentice-Hall International, 1995, pp. 3–28.
20. S. Oaks and H. Wong, *Jini in a nutshell*, O'Reilly and Associates, 2000.
21. G. Papadopoulos and F. Arbab, *Coordination models and languages*, Advances in Computers — The Engineering of Large Systems, vol. 46, 1998, pp. 329–400.
22. J. Rutten, *Elements of stream calculus (an extensive exercise in coinduction)*, Tech. report, CWI, Amsterdam, 2001.
23. J.-G. Schneider and O. Nierstrasz, *Components, scripts, glue*, Software Architectures - Advances and Applications (L. Barroca, J. Hall, and P. Hall, eds.), Springer-Verlag, 1999, pp. 13–25.
24. R. Siegel, *CORBA: Fundamentals and programming*, John Wiley & Sons Inc, 1997.
25. C. Szyperski, *Component software, beyond object-oriented programming*, Addison-Wesley, 1998.
26. M. Wermelinger and J. Fiadeiro, *Connectors for mobile programs*, IEEE Trans. on Software Eng. **24** (1998), no. 5, 331–341.