Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 181 (2007) 49–61
www.elsevier.com/locate/entcs

An Orchestrator for Dynamic Interconnection of Software Components

Marco Antonio Barbosa [1]

Departamento de Informática
Universidade do Minho
Braga, Portugal

Luís Soares Barbosa[2]

Departamento de Informática
Universidade do Minho
Braga, Portugal

Abstract

Composing and orchestrating software components is a fundamental concern in modern software engineering. This paper addresses the possibility of such orchestration being dynamic, in the sense that the structure of component's interconnection patterns can change at run-time. The envisaged approach extends previous work by the authors on the use of coalgebraic models for the specification of software connectors.

Keywords: Mobility, Software Connectors, Components.

1 Introduction

The increasing demand for complex and ubiquitous applications places new challenges to the way software is designed and developed. One of such challenges concerns the way in which an application deals with the dynamic reconfiguration of its components. Actually, components are no more static pieces of code assembled at compile time, but dynamic entities, often executing in different processing units, which interact only through well defined public interfaces. Component assembly is understood as interconnection of ports, declared in such interfaces, and more often than not such interconnection patterns change at runtime. This explains why, since the 1980's, *mobility* has become a buzzword both in academia and industry.

[1] Email: marco.antonio@di.uminho.pt
[2] Email: lsb@di.uminho.pt

doi:10.1016/j.entcs.2007.01.053

From a foundational point of view, the publication of Milner, Parrow and Walker original reports on the π-calculus [20], in 1992, was a fundamental milestone. Since then the topic became increasingly popular in both theorectical and applied research. It arises, for example, in connection with software architecture (*e.g.*, [7,22,21,13]), coordination models (*e.g.*, [17,16]) or programming languages (*e.g.*, [15,10]), just to name a few. But still, in the practice of software engineering, mobility remains hard to express and to be reasoned about.

Mobility is structurally associated with distribution, system's temporal evolution and dynamic creation or reconfiguration of processes, links or components' instances. In a previous set of publications [5,4] the authors proposed an approach to the specification of the coordination level of an application, based on independent, generic connectors modelled coalgebraically [25]. This approach is inspired by research on coordination languages [12,23] and favors strict component decoupling in order to support a looser inter-component dependency. Similarly to other approaches, like Reo [2] or Piccola [26,21], computation and coordination are clearly separated, communication becomes *anonymous* and component interconnection is externally controlled.

Such a model, however, assumed a fixed interconnection structure between the components and connectors involved, making the envisaged approach *static*. The present paper reports preliminary work on a possible extension of the framework in [4] to deal with dynamic reconfiguration. More precisely the model is extended by the explicit inclusion of a special connector — called the *orchestrator* — which plays the role of a connections manager. Such is achieved through a set of basic primitives to break or rebuild links between component's instances and connectors at run-time.

The proposed solution is essentially *operational*: it does not contribute for explaining the mathematics of mobility, but provides a possible way of dealing with dynamic reconfiguration within an exogenous coordination model. The basic requirement placed by the exogenous nature of the model is the absence of direct communication between component's instances. One of them, for example, may decide at some point to disconnect from a connector's port and, let us suppose, to send that port identifier to be part of a new connection. All it can do, however, is to post the port identifier through another connector's port. What will happen afterword, and in particular, which other component's instance, if any, will re-use such a port is not controlled by, in fact not even known to the original component. Each component's instance interfaces with the *glue* code, represented as a connector, and is not aware of the presence of other components equally connected to the same connector. That is why communication is anonymous and no direct reconfiguration orders (like 'link this to that') are not possible.

Notation. The paper resorts to standard mathematical notation typically used in model-oriented specification methods, like Vdm [14], Z [27] or B [1]. We adopt, however, 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

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

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

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 $f = \mathsf{ev} \cdot (\overline{f} \times C)$.

Of course all datatype constructions extend to functions. For example, notation $f + g : A + B \longrightarrow A' + B'$ denotes a function which, depending on the argument being of type A or B, applies f or g, returning as a result a value of types A' or B', respectively. Function composition is denoted as in $f \cdot g$ and function application by juxtaposition (as in $f\, x$). The identity function is represented as id.

Finally, we also assume the existence of a few basic sets, namely $\emptyset$, the empty set and $\mathbf{1}$, the (isomorphism class of the) singleton set. By convention $\perp$ is chosen as (the representative of) the unique element of $\mathbf{1}$. Also notice that, although in general elements of a set can only be accessed non deterministically, function the returns, in a quite deterministic way, the unique element of a singleton.

Paper Structure. The following section recalls from [4] the formal definition of software connectors and introduces a notion of *configuration* to refer to the joint behaviour of component's instances and the connector linking them. Section 3 reports the main contribution of the paper, explaining in which sense the model in [4] can be adapted to cope with dynamic reconfiguration and paving the way to the formal definition of the *orchestrator* connector given in section 3. Section 4 presents an example. The paper concludes in section 5 presenting a number of relevant issues for future work. It reports, in particular, on a prototype implementation of this model in Haskell [8].

2 Connectors, Components and Configurations

2.1 Software Connectors

In the model proposed in [4], the specification of a software connector is given by an interface which aggregates a number of *ports*. Each port has an associated operation which regulates its behaviour, by encoding the port reaction to a message

crossing the connector's boundary. Let U be the type of the connector's internal state space and $\mathbb{M}$ a generic type for messages. There are three kinds of ports[3] with the following signatures:

$$\mathsf{post} \ : \ U \longrightarrow U^{\mathbb{M}} \tag{3}$$

$$\mathsf{get} \ : \ U \longrightarrow U \times (\mathbb{M}+1) \tag{4}$$

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

Notice that Ports have an interaction polarity, either *input* or *output*, and are uniquely identified. There is only one sort of input port, whereas output ports may be either destructive, in the sense that a data value is deleted from the port once read, or nondestructive. In detail, the intuition is as follows:

- post is an input operation analogous to a write operation in conventional programming languages (see *e.g.*, [2]). 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.

In all cases, at creation time, ports are uniquely labelled with a distinguished identifier taken from an enumerable set $\mathbb{P}$. The existence of (global) unique port identifiers, together with the possibility of them being communicated through channels, makes dynamic reconfiguration possible. Actually, and this is a fundamental difference with respect to the authors previous work as documented in [5,4], a message of type $\mathbb{M}$ may be either a data value (of a generic type $\mathbb{D}$) or a port identifier ($\mathbb{P}$). Formally, the set of messages is defined as $\mathbb{M} = \mathbb{D} + \mathbb{P}$, where $+$ is, as explained above, datatype sum.

2.2 The General Case

A software connector is specified by an interface which aggregates a number of P post, G get and R read ports. Such an aggregation leads to the following general definition of a connector, as a coalgebra [25] over state space U

$$c \ = \ \langle u \in U, \langle \gamma_c, \rho_c \rangle : U \longrightarrow (U \times (\mathbb{M}+1))^{P \times \mathbb{M} + G} \times (\mathbb{M}+1)^R \rangle \tag{6}$$

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

$$\rho_c : \ U \longrightarrow (\mathbb{M}+1) \times (\mathbb{M}+1) \times \ldots \times (\mathbb{M}+1) \tag{7}$$

[3] named according to the *client's* (*i.e.*, the connector's environment) point of view.

and, γ_c collects ports of type post or get, which may be required to perform state updates. Actually the type of the codomain of the coalgebra in (6) can be rewritten as

$$U \;=\; (\sum_{i\in P} U^{\mathbb{M}} + \sum_{j\in G} U \times (\mathbb{M}+1)) \times \prod_{k\in R} (\mathbb{M}+1) \tag{8}$$

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

As an example of a connector's definition, consider the construction of a binary connector. 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. For example, the aggregation of a post and a read ports leads to

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

On the other hand, replacing the read port above by a get forces an additive aggregation to avoid the possibility of simultaneous updates leading to

$$c \;=\; \langle u \in U, \gamma_c : U \to (U \times (\mathbb{M}+1))^{\mathbb{M}+\mathbf{1}}\rangle \tag{10}$$

where [4]

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

Channels of different kinds are connectors of this type. The *synchronous* and the *asynchronous* channels are the most well-known examples. The reader is referred to [4] for a full account of the envisaged connector's semantics and construction.

2.3 Components and Configurations

In an exogenous coordination model component instances are always regarded as *black boxes* (see, e.g., [3]). All that is assumed to be known about them are

- the port interface signature, *i.e.*, the identifier and polarity of each of its ports
- a specification of the *interface behaviour*, which defines what is called here the *component's usage*.

This is given by a process algebra-like expression and intended to define the activation pattern of the component interface. For example,

$$beh(C) \;=\; (in.in.out)^*$$

establish that the port activation pattern for component instance C requires two activations of port *in* before an activation of port *out*. The notation used is based on

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

Ccs [18] over a set *Act* of actions, each action corresponding to a port activation [5]. Differently from Ccs, however, actions come equipped with a *co-occurrence* operator $\|$ — action $a\|b$ stands for the simultaneous occurrence of both a and b. Syntax is as follows:

$$B ::= \mathbf{0} \mid Act.B \mid B + B \mid B|B \mid B\backslash K \mid [\sigma]B \mid B^*$$

where, $\cdot$ is the *prefix* operator, $+$ and $|$ denote *non deterministic choice* and *parallel* composition, respectively. Notation $B\backslash K$ represents *restriction* to a set K of actions, $[\sigma]$ stands for action *renaming* in accordance with substitution σ, and $*$ denotes *iteration*.

As described above, a connector is *internally* specified as a coalgebra built by port aggregation. Its external behaviour, however, is also given by a process algebra expression. For example, behaviour of a synchronous channel with port *in* and *out* is given by $(in\|out)^*$ whereas the asynchronous case is specified by $(in.out)^*$.

Component instances never interact with each other directly, but always through the connector network. Actually they are not even aware of each other's presence. The whole system is described by a number of component instances and a connector built from elementary connectors (like, e.g., synchronous and asynchronous channels) using a connector's algebra formally described in [4]. This algebra includes a parallel aggregation ($c_1 \boxtimes c_2$) and a feedback mechanism ($C\ \circlearrowleft$) which links selectively ports of opposite polarity in C. The joint behaviour of connectors and component instances in a particular setting is called a *configuration*. This describes, in particular, the actual connections between ports. The semantics of configurations, which is parametric on an interaction discipline along the lines of [24] is reported in [6].

3 The Orchestrator Connector

3.1 The Orchestrator

Central to our approach is the presence of a particular coordination connector, called the *orchestrator*, whose role is to manage the interactions among all other elements in the architectural network. In a sense, the *orchestrator* corresponds to an intermediary layer between components and ordinary connectors, *i.e.*, it acts as a bridge among components and connectors.

The *orchestrator* is a listener permanently attentive to the flow of messages which do not contain data but port identifiers, instead. Such messages will be understood as an order for control transfer. Actually, the orchestrator is triggered whenever a message with a port identifier *arrives* at a port of a particular component instance. This means that interception is made on the execution of a read or get operation. At this point, it intercepts the message, and re-organizes the overall network connections according to some *reconfiguration script*. Notice, however,

[5] A somewhat more expressive notation for behavioural specifications is suggested in [24,6], which is however not essential for the purposes of this paper.

that neither component's instances nor the connectors in the network are aware of its presence.

To fulfill its role the *orchestrator* is defined as a coalgebra over state space U specified as datatype

$$U \ : \ \mathbb{P} \times Cmp \times \mathcal{P}(\mathbb{P} \times \mathbb{P}) \tag{11}$$

where $\mathbb{P}$ is the set of *port identifiers* known to the orchestrator, $\mathcal{P}(\mathbb{P} \times \mathbb{P})$ is a set of active *connections* and Cmp is a *component manager* defined as function

$$Cmp = (\mathcal{P}\mathbb{P})^{cmpId} \tag{12}$$

which associates to each component instance the set of its ports. There is an obvious type invariant associated to U stating that all connections are point-to-point:

$$\mathsf{inv}\, u \ = \ \forall_{p \in \pi_1 u} \,.\ \mathsf{card}\,\{c \in \pi_3 u \mid \pi_1 c = p \ \vee \ \pi_2 c = p\} \ \leq 1 \tag{13}$$

3.2 Specification of the Orchestrator Operations

The underling operations are defined as follows

- *Get Connection* (getCon): This operation takes a port identifier and, if such an identifier is part of a connection, returns the corresponding end point. Formally,

$$\begin{aligned}
\mathsf{getCon} \ &: U \times \mathbb{P} \to \mathbb{P} + 1 \\
\mathsf{getCon}(u,p) \ &\triangleq \\
&\quad \mathit{let} \\
&\qquad e \ = \ \{\pi_2 c |\ c \in \pi_3 u \ \wedge \ \pi_1 c = p\} \cup \{\pi_1 c |\ c \in \pi_3 u \ \wedge \ \pi_2 c = p\} \\
&\quad \mathit{in} \\
&\qquad (e \neq \emptyset \ \to \ \iota_1\, \mathsf{the}(e),\ \iota_2\, \bot)
\end{aligned}$$

- *Disconnection* (*disCon*): This operation updates the connector's state space by deleting an existing connection.

$$\begin{aligned}
\mathsf{disCon} \ &: \ U \times (\mathbb{P} \times \mathbb{P}) \to U \\
\mathsf{disCon}(u,(p,p')) \ &\triangleq \ (\pi_1 u, \pi_2 u, (\pi_3 u) \backslash \{(p,p')\})
\end{aligned}$$

- *Add Port* (*addPort*): This operation updates the connector's state space by adding a new available port.

$$\begin{aligned}
\mathsf{addPort} \ &: \ U \times \mathbb{P} \to U \\
\mathsf{addPort}(u,p) \ &\triangleq \ (\{p\} \cup \pi_1 u, \pi_2 u, \pi_3 u)
\end{aligned}$$

- *Available Ports (avPort)*: This operation searches for ports in a given component

instance available for connection.

$$
\begin{aligned}
\mathsf{avPort} \;&:\; U \times cmpId \to \mathcal{P}(\mathbb{P}) \\
\mathsf{avPort}(u,i) \;&\triangleq \\
&\quad let \\
&\qquad \mathsf{used} \;=\; \mathcal{P}(\pi_1)\,(\pi_3 u) \;\cup\; \mathcal{P}(\pi_2)\,(\pi_3 u) \\
&\quad in \\
&\qquad (\pi_3 u)\,i \;\cap\; \mathsf{used}
\end{aligned}
$$

- *Make Connection* (*mkCon*): This operation aggregates a new connection to the orchestrator's state space.

$$
\begin{aligned}
\mathsf{mkCon} \;&:\; U \times (\mathbb{P} \times \mathbb{P}) \to U \\
\mathsf{mkCon}(u,(p,p')) \;&\triangleq\; (\pi_1 u, \pi_2 u, \{(p,p')\} \cup (\pi_3 u))
\end{aligned}
$$

3.3 Coordination Patterns

The set of primitives above are used to build the above mentioned *reconfiguration scripts* which constitute the orchestrator reaction to the interception of (incoming) messages. The idea is that the orchestrator's behaviour is parameterized by such scripts, which, given their role in the model, will be also referred to as *coordination pattern*. Let us consider one of such patterns to illustrate how they can be specified in terms of the orchestrator primitives.

The intuition is as follows:

> On interception of a message containing a port identifier m arriving to port p, the orchestrator identifies the component instance to which p belongs and tries to find another port in it to connect to m. In case m was previously part of another connection, such connection has to be traced and broken.

Formally,

$$
\begin{aligned}
\mathsf{pattern}_1(u,p,m) \;&\triangleq \\
&\quad let \\
&\qquad ap \;=\; \mathsf{avPort}(u, \mathsf{owner}(u,p)) \\
&\quad in \\
&\qquad (ap \neq \emptyset \;\rightarrow\; let\ r \in ap\ in\ \mathsf{reconf}(u,p,m,r)\,,\; u)
\end{aligned}
$$

where

$$\begin{array}{l}\mathsf{reconf}(u,p,m,r) \triangleq \\ \quad let \\ \quad\quad x = \mathsf{getCon}(u,m) \\ \quad in \\ \quad\quad (x = \iota_1(m') \rightarrow \mathsf{mkCon}(\mathsf{disCon}(u,(m,m')),(m,r)), \mathsf{mkCon}(u,(m,r)))\end{array}$$

and function owner inspects the *component manager* in the orchestrator's state to return the identifier of the component instance to which a particular port belongs. Formally,

$$\mathsf{owner}(u,p) \triangleq \mathsf{the}\,\{c \in cmpId |\ p \in (\pi_2 u)\, c\}$$

Note that in this coordination pattern if there is no port available for the new connection, the configuration if not changed. This is not, however, the only possibility. Reasonable alternatives would be

- to disconnect port m in any case
- or to suspend until the a port becomes available for connection in $\mathsf{owner}(u,p)$.

Such alternatives can also be encoded as *coordinating patterns* to act as a parameter to the orchestrator.

4 Example

For illustration purposes let us consider how to model, in the framework outlined in the previous sections a variation of the example presented in [19].

In this case we shall consider a *wireless network* where a notebook (component *Client*) is connected to a network which has two servers ($Base_1$ and $Base_2$). These two servers are connected to each other and the client is connected to one of them.

In the initial configuration of the system the *Client* is communicating with the server $Base_1$ according to the Fig 1. The *Client* may permanently communicate with the network through its input port c_i and output port c_o. Such a behaviour is captured by

$$beh(Client) = (c_i + c_o)^*$$

$Base_1$ is permanently communicating with $Base_2$, through its output port $b_{1.3}$ and input port $b_{1.4}$. In such a system configuration $Base_1$ is also communicating with the *Client* using the output port $b_{1.1}$ and the input port $b_{1.2}$. This behaviour is given by,

$$beh(Base_1) = (b_{1.1} + b_{1.2} + b_{1.3} + b_{1.4})^*$$

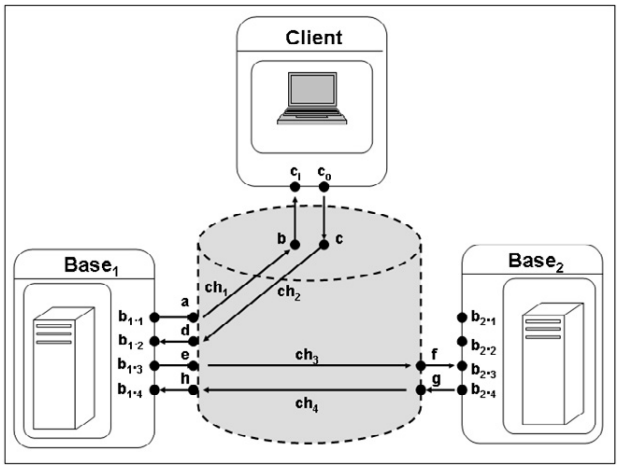

Fig. 1. The initial configuration.

$Base_2$ communicates with $Base_1$ though its input port $b_{2.3}$ and output port $b_{2.4}$. $Base_2$ also has an input port $b_{2.1}$ and an output port $b_{2.2}$. In this case both ports are available and disconnected from the connectors network. The behaviour of $Base_2$ is given by,

$$beh(Base_2) \;=\; (b_{2.3} \;+\; b_{2.4})^*$$

The components involved in the network are interconnected by a connector made of the four synchronous channels depicted in Fig 1. Its behaviour $beh(C)$ is obtained by the parallel composition of each channel. The reader is referred to [6] for details. For the moment it is enough to point out that, as usual, the semantics of parallel composition is given in terms of both interleaving and synchronous product.

The whole system is specified by configuration $conf$ whose behaviour is

$$\begin{aligned} beh(conf) \;=\; & beh(Client)[b/c_i, c/c_o] \mid beh(C) \\ \mid\; & beh(Base_1)[a/b_{1.1},\; d/b_{1.2},\; e/b_{1.3},\; h/b_{1.4}] \\ \mid\; & beh(Base_2)[f/b_{2.3},\; g/b_{2.4}] \end{aligned}$$

Note that the renaming operation connects the components ports to the connectors ports.

Now, let us consider that the user moves and the signal becomes weak. The server $Base_1$ communicates with the server $Base_2$ and sends the port identifiers in order for $Base_2$ to provide the service. The Fig 2 represents the result of such an operation.

After the *orchestrator* has intercepted the message the behaviour of the configuration becomes

$$\begin{aligned} beh(conf') \;=\; & beh(Client)[b/c_i, c/c_o] \mid beh(C) \\ \mid\; & beh(Base_1)[e/b_{1.3}, h/b_{1.4}] \\ \mid\; & beh(Base_2)[d/b_{2.1}, a/b_{2.2}, f/b_{2.3}, g/b_{2.4}] \end{aligned}$$

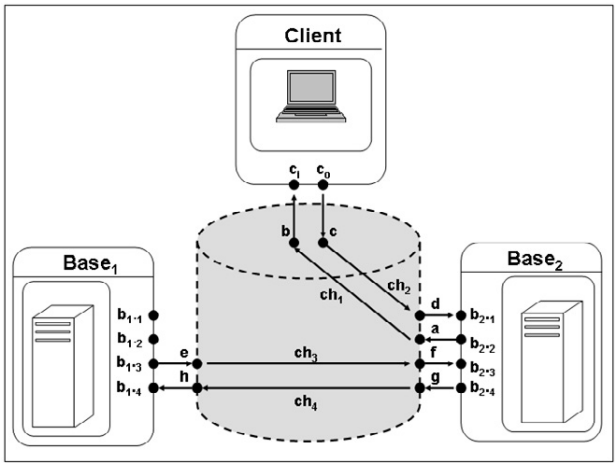

Fig. 2. The final configuration.

Let us now focus on the *orchestrator* role. Suppose it is parameterized with $\mathsf{pattern}_1$ above and let its initial state be $u =< p, cmp, con >$, where

$$\begin{aligned} p &= \{c_i, c_o, b_{1.1}, b_{1.2}, b_{1.3}, b_{1.4}, b_{2.1}, b_{2.2}, b_{2.3}, b_{2.4}\} \\ cmp &= \{(Client, \{c_i, c_o\}), (Base_1, \{b_{1.1}, b_{1.2}, b_{1.3}, b_{1.4}\}), (Base_2, \{b_{2.1}, b_{2.2}, b_{2.3}, b_{2.4}\})\} \\ con &= \{(a, b), (c, d), (e, f), (g, h)\} \end{aligned}$$

Suppose, now, the following situation occurs: $Base_1$ sends port identifier a of channel ch_1 to $Base_2$ through the connector C. The *orchestrator* captures such a message and starts the script defined in $\mathsf{pattern}_1$, as follows.

- With avPort the *orchestrator* selects port $b_{2.2}$ as an alternative to connect to a.
- With operation getCon it obtains port $b_{1.1}$, which was previously connected to a.
- As such a port has a current connection, the following step is to break it with disCon.
- Finally, the new connection, linking a to $b_{2.2}$ is completed.

5 Conclusions and Future Work

This paper discussed an extension to a formalization of software connectors reported in [4], which adopts a coordination oriented approach to support looser levels of inter-component dependency. The framework supports dynamic reconfiguration of connections through the action of a special connector (abstracting a whole level of middleware) which manages the active possibilities of communication.

The possibility of dynamic configuration of connections arises in this model from two basic assumptions: *(a)* ports have unique identifiers which can be exchanged in messages, *(b)* there is a special connector — the *orchestrator* — to manage all active connections in the network. With them mobility can be achieved in the classical *name-passing* style typical of process algebras of the π-calculus family [19].

A lot of work remains to be done. In particular this approach should be compared with formal approaches to *dynamically re-configurable* architectures, such as, for example, [11] or [28]. Our main current concerns, however, include the full development of the model and associated calculus, as well as its application to realistic case-studies.

We are also currently working on a prototype implementation of this model as a HASKELL library, on top of which experimentation can proceed. In particular, we intend to build a repository of *coordination patterns* to be used as parameters to the orchestrator, in order to capture a number of typical coordination situations.

Acknowledgement

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] J. R. Abrial. *The* B *Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] 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.

[3] F. Arbab. Reo: a channel–based coordination model for component composition. *Mathematical Structures in Comp. Sci.*, 14(3):329–366, 2004.

[4] M. Barbosa and L. Barbosa. Specifying software connectors. In K. Araki and Z. Liu, editors, *Proc. First International Colloquim on Theoretical Aspects of Computing (ICTAC'04), Guiyang, China*, pages 53–68. Springer Lect. Notes Comp. Sci. (3407), 2004.

[5] M. A. Barbosa and L. S. Barbosa. A relational model for component interconnection. 10(7):808–823, 2004.

[6] M. A. Barbosa and L. S. Barbosa. How do components connect? the semantics of configurations. Technical Report DI-PURe-06.03.01, Universidade do Minho, Braga, Portugal, March 2006.

[7] K. Bergner, R. Grosu, A. Rausch, A. Schmidt, P. Scholz, and M. Broy. Focusing on mobility. In *HICSS*, 1999.

[8] R. Bird. *Functional Programming Using Haskell*. Series in Computer Science. Prentice-Hall International, 1998.

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

[10] A. R. D. Bois, P. Trinder, and H. Loidl. mHaskell: Mobile computation in a purely functional language. *Journal of Universal Computer Science*, 11(7):1234–1254, 2005.

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

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

[13] V. Gruhn and C. Schäfer. An architecture description language for mobile distributed systems. In R. M. Flavio Oquendo, Brian Warboys, editor, *Software Architecture - Proceedings of the First European Workshop, EWSA 2004*, pages 212–218. Springer-Verlag, 2004.

[14] C. B. Jones. *Systematic Software Development Using* VDM. Series in Computer Science. Prentice-Hall International, 1986.

[15] Z. D. Kirli. *Mobile Computation with Functions*, volume 5 of *Advances in Information Security*. Springer, 2002.

[16] A. Lopes and L. Fiadeiro. On how distribution and mobility interfere with coordination. In *Proc. of WADT*, pages 343–358. Springer Lect. Notes Comp. Sci (2755), 2002.

[17] M. Lumpe. *A π-calculus Based Approach to Software Composition*. PhD thesis, University of Bern, January 1999.

[18] R. Milner. *Communication and Concurrency*. Series in Computer Science. Prentice-Hall International, 1989.

[19] R. Milner. *Communicating and Mobile Processes: the π-Calculus*. Cambridge University Press, 1999.

[20] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes (parts I and II). *Information and Computation*, 100(1):1–77, 1992.

[21] 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.

[22] F. Oquendo. π-adl: an architecture description language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures. *SIGSOFT Softw. Eng. Notes*, 29(3):1–14, 2004.

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

[24] P. R. Ribeiro, M. A. Barbosa, and L. S. Barbosa. Generic process algebra: A programming challenge. In *Proc. 10th Brazilian Symposium on Programming Languages*, Itatiaia, Brasil, 2006.

[25] 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).

[26] 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.

[27] J. M. Spivey. *The* Z *Notation: A Reference Manual (2nd ed)*. Series in Computer Science. Prentice-Hall International, 1992.

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