Reconfiguration mechanisms for service coordination*

Nuno Oliveira and Luís S. Barbosa

HASLab/INESC TEC & Universidade do Minho
`{nunooliveira,lsb}@di.uminho.pt`

Abstract. Models for exogenous coordination provide powerful *glue-code*, in the form of software connectors, to express interaction protocols between services in distributed applications. Connector reconfiguration mechanisms play, in this setting, a major role to deal with change and adaptation of interaction protocols. This paper introduces a model for connector reconfiguration, based on a collection of primitives as well as a language to specify connectors and their reconfigurations.

1 Introduction

The purpose of a service-oriented architecture (SOA)[10,11] is to address requirements of loosely coupled and protocol-independent distributed systems, where software resources are packaged as self-contained *services* providing well-defined functionality through publicly available interfaces. The architecture describes their interaction, ensuring, at the same time, that each of them executes independently of the context or internal state of the others.

Over the years a multitude of technologies and standards [1] have been proposed for describing and orchestrating web services, publish and discover their interfaces and enforce certain levels of security and QoS parameters. Either to respond to sudden and significative changes in context or performance levels, or simply to adapt to evolving requirements, some degree of *adaptability* or *reconfigurability* is typically required from a service-oriented architecture. By a (dynamic) reconfiguration we mean a process of adapting the architectural current configuration, once the system is deployed and without stopping it [14], so that it may evolve according to some (emergent) requirements [12] or change of context.

Reconfigurations applied to a SOA may be regarded from two different point of views. From one of them, they target individual services [21]. In particular, such reconfigurations are concerned with dynamic update of services, substitution of a service by another with compatible interfaces (but not necessarily the

---

* This work is funded by ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT (Portuguese Foundation for Science and Technology) within project FCOMP-01-0124-FEDER-010047. The first author is also supported by an Individual Doctoral Grant with reference number SFRH/BD/71475/2010

same behaviour) or even their plain removal. Such reconfigurations are usually triggered by external stimulus [19,13,17,18,23]. Form another point of view, a reconfiguration is entirely decided by the system itself and targets the way components or services interact with each other, as well as the internal QoS levels measured along such interactions. In particular, such reconfigurations deal with substitution, addition or removal of communications channels, moving communication interfaces from a service to another or rearranging a complex interaction structure.

This paper studies reconfiguration mechanisms for the service interaction layer in SOA. Adopting a coordination-based view of interaction [20], the model proposed here represents the 'gluing-code' by a graph of *channels* whose nodes represent interaction points and edges are labelled with channel identifiers and types. A channel abstracts a point-to-point communication device with a unique identifier, a specified behaviour and two ends. It allows for data flow by accepting data on a *source* end and dispensing it from a *sink* end. We call such a graph a *coordination pattern.* A subset of its nodes are intended to be plugged to concrete services, forming the pattern interface.

To keep things concrete, we assume channels in a coordination pattern are described in a specific coordination model, that of Reo [3,2]. Actually, this choice is not essential: the reconfiguration mechanisms are directly defined over the graph and concern only its topology. Only when one intends to reason about the system's behaviour or compare the behavioural effect of a reconfiguration, does the specific semantics of the underlying coordination model become relevant. Such is not addressed, however, in this paper.

Coordination patterns are introduced in Section 4 and instantiated in the context of the Reo coordination model. Section 3 discusses reconfigurations, formally defining a collection of primitives. It is shown how the latter can be combined to yield 'big-step' reconfiguration patterns which manipulate significative parts of a pattern structure. The CooPLa language is introduced in Section 4 as an executable notation for specifying both coordination and reconfiguration patterns. Reconfiguration mechanisms are illustrated through a detailed example in Section 5. Section 6 concludes the paper.

2 Coordination patterns

A pattern is an effective, easy to learn, repeatable and proven method that may be applied recurrently to solve common problems [10]. They are common in several domains of Software Engineering, namely in SOA [22] and business process [25].

Similarly, in this paper, a coordination pattern encodes a reusable solution for an architectural (coordination) problem in the form of a specific sort of *interaction* between the system constituents. A solution for an architectural problem is, therefore, the description of interaction properly designed to meet a set of requirements or constraints. It is reflected in a coordination protocol, which acts as *glue-code* for the components or services interacting within the system.

Formally, a coordination pattern is presented by a graph of *channels* whose nodes represent interaction points and edges are labelled with channel identifiers and types. As explained in the Introduction, we adopt here the Reo framework [2], in order to give a concrete illustration of our approach.

Let $\mathcal{N}ame$ and $\mathcal{N}ode$ denote, respectively, a set of unique names and a set of nodes associated either with coordination patterns or channels. A node can also be seen as an interaction *port*. It is assumed the following set of primitive types of channels (see Fig. 1) with the usual Reo [3,2]semantics.

$$\mathcal{T}ype \overset{\text{def}}{=} \{\mathsf{sync}, \mathsf{lossy}, \mathsf{fifo_f}, \mathsf{fifo_e}, \mathsf{drain}\}$$

Each channel has exactly two ends and are, normally, directed (with a source and a sink end) but Reo also accepts undirected channels (*i.e.,* channels with two ends of the same sort). Channel ends form the nodes of coordination patterns. A node may be of three distinct types: (*i*) source node, if it connects only source channel ends; (*ii*) sink node, if it connects only sink channel ends and (*iii*) mixed node, if it connects both source and sink nodes. Fig. 1 recalls the basic channels used in Reo through the composition of which complex coordination schemes can be defined. The sync channel transmits data from one end to another whenever there is a request at both ends synchronously, otherwise one request shall wait for the other. The lossy channel behaves likewise, but data may be lost whenever a request at the source end is not matched by another one at the sink end. Differently, a fifo channel has a buffering capacity of one memory positions, therefore allowing for asynchronous occurrence of I/O requests. Qualifier e or f refers to the channel internal state (either *empty* or *full*). Finally, the synchronous drain channel accepts data synchronously at both ends and loses it.
We use $\mathcal{P}$ to denote the set of all coordination patterns. A *coordination pattern* is defined as follows:

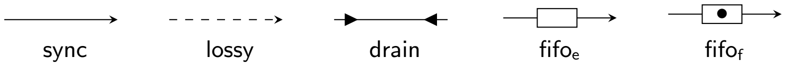


Fig. 1. Primitive Reo channels.

Definition 1 (Coordination pattern). *A coordination pattern is a triple*

$$\rho \overset{\text{def}}{=} \langle I, O, R \rangle$$

- $R \subseteq \mathcal{N}ode \times \mathcal{N}ame \times \mathcal{T}ype \times \mathcal{N}ode$ *is a graph on connector ends whose edges are labelled with instances of primitive channels, denoted by a channel identifier (of type* $\mathcal{N}ame$*) and a type (of type* $\mathcal{T}ype$*);*
- $I, O \subseteq \mathcal{N}ode$ *are the sets of source and sink ends in graph* R*, corresponding to the set of input and output ports in the coordination pattern, respectively.*

Clearly, every channel instance gives rise to a coordination pattern. For example pattern

$$\rho_s = \langle \{a\}, \{b\}, \{\langle a, sc, \mathsf{sync}, b\rangle\}\rangle$$

corresponds to a single synchronous channel, identified by sc, linking an input port a to an output port b. Similarly, plugging to its output port two lossy channels yields a lossy broadcaster which replicates data arriving at a, if there exist pending reading requests at d and e:

$$\rho_b = \langle \{a\}, \{d, e\}, \{\langle a, sc, \mathsf{sync}, b\rangle, \langle b, l_1, \mathsf{lossy}, d\rangle, \langle b, l_2, \mathsf{lossy}, e\rangle\}\rangle$$

A drain, on the other hand, has two source, but no sink, ends. Therefore, a pattern formed by an instance of a drain channel resorts to a special end $\perp \in \mathcal{N}ode$ which intuitively represents absence of data flow. Thus, and for example,

$$\rho_d = \langle \{a, b\}, \emptyset, \{\langle a, ds, \mathsf{drain}, \perp\rangle, \langle b, ds, \mathsf{drain}, \perp\rangle\}\rangle$$

As a matter of fact, invariants to avoid ill-formed coordination patterns, are required. For instance (i) the $\perp$ ports can never be connected to other ports (ii) a name may only be associated to two different ports and a unique channel type (notice the veracity of this also in the drain example) or (iii) only a single channel is allowed to connect two consecutive nodes. We assume the existence of such invariants, and do not address them in this paper.

Notice, however, that the defintion of coordination pattern may be relaxed. Instead of regarding it as a triple, one may drop the first two components (I and O), since these can be *extracted* from the component R—the graph—which is preserved.

A *port* in a coordination pattern is a channel end not connected (to any other channel). In the context of the R component, a node is identified as a port if it only appears as either the first or the fourth component in every elements of R. Formally, for $\rho \in \mathcal{P}$, with $\rho = R$

$$\begin{array}{lcl} I(\rho) & \overset{\text{def}}{=} & \{\pi_1(e) \mid e \in R \wedge in(\pi_1(e), R)\} \\ in(x, S) & \overset{\text{def}}{=} & \nexists_{e \in S} \ . \ \pi_4(e) = x \end{array}$$

defines the set I (of input ports) of the coordination pattern. Dually,

$$\begin{array}{lcl} O(\rho) & \overset{\text{def}}{=} & \{\pi_4(e) \mid e \in R \wedge in(\pi_4(e), R)\} \\ in(x, S) & \overset{\text{def}}{=} & \nexists_{e \in S} \ . \ \pi_1(e) = x \end{array}$$

defines the set O (of output ports) of the coordination pattern.

In the remaining of this document, input and output ports are accessed via the above operations. ρ (possibly with indexes) is used to refer to a coordination pattern and its component R. This relaxation allows for a smooth introduction to the reconfiguration mechanisms.

3 Architectural reconfigurations

This section discusses reconfigurations of coordination patterns. We take a rather broad view of what a reconfiguration is: any transformation obtained through a sequence of elementary operations, described below, is qualified as a reconfiguration. Our aim is to build a framework in which such transformations can be defined and the effect of their application to a specific pattern assessed. Later, one may restrict this set, for example by ruling out transformations which do not preserve the pattern input-output interface or fail to lead to patterns with a behaviour which simulates (or bisimulates) the original one. Such considerations, however, require the assumption of a specific semantics for coordination patterns, easily built from any Reo semantic model, but which lies outside the more 'syntactic' scope of this paper.

Definition 2 (Reconfiguration). *A reconfiguration is a sequence* r *of operations* $\langle o_0, o_1, \ldots, o_n \rangle$*, where each* o_i *belongs to the set*

$$\mathcal{O}p \overset{\text{def}}{=} \{\mathsf{par},\ \mathsf{join},\ \mathsf{split},\ \mathsf{remove}\}$$

of elementary reconfigurations, specified below. The application of a reconfiguration r *to a pattern* ρ *yields a new pattern and is denoted by* $\rho \bullet r$*.*

3.1 Primitive reconfigurations

Let us start by defining the set of elementary reconfigurations of a coordination pattern. The simplest reconfiguration is *juxtaposition*. Intuitively, it sets two coordination patterns in parallel without creating any connection between them. Formally,

Definition 3 (The par operation). *Let* ρ_1 *and* ρ_2 *be two coordination patterns. Then,*

$$\rho_1 \bullet\ \mathsf{par}(\rho_2) = \rho_1 \uplus \rho_2$$

where $\uplus$ *is set disjoint union.*

The par operation assumes disjunction of nodes and channel identifiers in the patterns to be joined. This is assumed without loss of generality, because formally a disjoint union of all identifiers is previously made.

The second elementary reconfiguration is join. Intuitively, it creates a new node j that superposes all nodes in a given set P. This operation adds fresh node j as a new input or output port if all the nodes in P are, respectively, input or output ports in ρ. Formally,

Definition 4 (The join operation). *Let* $\rho \in \mathcal{P}$*,* $P \subseteq \mathcal{N}ode$ *and* $j \in \mathcal{N}ode$*. Then,*

$$\rho \bullet\ \mathsf{join}(P, j) = \rho'$$

– $\rho' = 2^{jn_{P,j}}(\rho)$, *with*

$$jn_{P,j}\langle q, id, t, s\rangle = \langle (q \in P \rightarrow j,\, q), id, t, (s \in P \rightarrow j,\, s)\rangle$$

The notation $(\phi \rightarrow s,\, t)$ corresponds to McCarthy's conditional, returning s or t if predicate ϕ is true or false, respectively. Also note that the power set of a set A is denoted by 2^A and, for a function f from A to B, $2^f(X) = \{f\,x \mid x \in X\}$.

The join operation has two pre-conditions. Clearly, node j must be a fresh name in ρ, or at least, a node within set P. Additionally, every node in P shall exist as a node of ρ. Formally, $\forall_{p\in P}.\, \exists_{e\in\rho}.\, \pi_1(e) = p \vee \pi_4(e) = p$.

The dual to join is the split operation which takes a node p in a pattern and breaks connections, separating all channel ends coincident in p. Technically this is achieved by renaming every occurrence of node p in all $e \in R$ to a fresh name $a.p$ or $p.a$ depending on whether p appears as a sink node in e and a is the corresponding source end, or the other way round. Thus,

Definition 5 (The split operation). *Let $\rho \in \mathcal{P}$, and $p \in \mathcal{N}ode$. Then,*

$$\rho \bullet \ \mathsf{split}(p) = \rho'$$

– $\rho' = 2^{sp_p}(\rho)$, *with*

$$sp_p\langle q, ch, t, s\rangle = \langle ((q = p) \rightarrow p.s,\, q), ch, t, ((s = p) \rightarrow q.p,\, s)\rangle$$

Finally, the remove operation removes a channel from a coordination pattern, if it exists.

Definition 6 (The remove operation). *Let $\rho \in \mathcal{P}$, and $ch \in \mathcal{N}ame$. Then,*

$$\rho \bullet \ \mathsf{remove}(ch) = \rho \setminus D$$

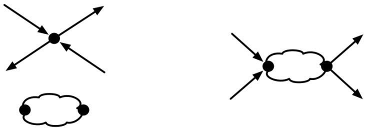
where, $D = \{e \mid e \in R \wedge \pi_2(e) = ch\}$

3.2 Reconfiguration patterns

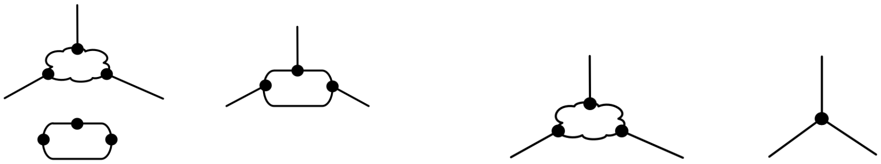
Practice and experience in software architecture inspire the definition of patterns for reconfiguring architectures. As stated in the Introduction, the focus of traditional reconfiguration is set on the replacement of individual components, rather than on the interaction protocols. The pattern presented here, on the other hand, are focussed on the latter, but still at this lower-level the interest is in defining 'big step' reconfigurations, by replacing simultaneously significant parts of a pattern. Fig. 2 sums up the set of such reconfiguration patterns we have found useful in practice.

The first one removes from a pattern a whole set of channels, applying the remove primitive systematically,

Remove pattern *Overlap pattern*



Insert pattern



Replace pattern *Implode pattern*

Fig. 2. Reconfiguration patterns

Definition 7 (The removeP **pattern).** *Let* $\rho \in \mathcal{P}$ *and* Cs *be a set of channels to remove. Then,*

$$\rho \bullet \mathsf{removeP}(Cs) = rS(\rho, Cs)$$

where

$$\begin{aligned} &rS(\rho, \emptyset) = \rho \\ &rS(\rho, Cs) = \text{let } \ c \in Cs \ \text{ in } \ rS(\rho \bullet \mathsf{remove}(c), Cs \setminus \{c\}) \end{aligned}$$

Another common reconfiguration overlaps two patterns by joining nodes from both of them. This is specified by a set of triples indicating which nodes are to be overlapped and a fresh name for the result. Formally,

Definition 8 (The overlapP **pattern).** *Let* $\rho, \rho_r \in \mathcal{P}$ *and* X *be a set of triples of nodes, where the first component is a node of* ρ*, the second one is a node of* ρ_r *and the third is a fresh node in both coordination patterns. Then,*

$$\rho \bullet \mathsf{overlapP}(\rho_r, X) = rO(\rho \bullet par(\rho_r), X)$$

where

$$\begin{aligned} rO(\rho, \emptyset) &= \rho \\ rO(\rho, X) &= \ \text{let } \ e_i \in X, E_i = \{\pi_1(e_i), \pi_2(e_i)\}, \\ &\quad\ \text{in } \ rO(\rho \bullet \mathsf{join}(E_i, \pi_3(e_i)), X \setminus \{e_i\}) \end{aligned}$$

The insertP pattern puts both patterns side by side, uses split to make room for a new pattern to be added, as shown in Fig. 2, and join to re-build connections. Formally,

Definition 9 (The $\mathsf{insertP}$ pattern). *Let $\rho, \rho_r \in \mathcal{P}$ and $n, m_i, m_o, j_1, j_2 \in \mathcal{N}ode$, where n is a node of ρ, m_i, m_o are input and output nodes, respectively, of ρ_r and j_1, j_2 are fresh nodes. Then,*

$$
\begin{array}{rl}
\rho \bullet \mathsf{insertP}(\rho_r, n, m_i, m_o, j_1, j_2) = \text{let} & \rho_1 = \rho \bullet \mathsf{par}(\rho_r) \\
 & \rho_2 = \rho_1 \bullet \mathsf{split}(n) \\
 & I_{sp} = I(\rho_2) \setminus I(\rho_1) \\
 & O_{sp} = O(\rho_2) \setminus O(\rho_1) \\
 & \rho_3 = \rho_2 \bullet \mathsf{join}(O_{sp} \cup \{m_i\}, j_1) \\
\text{in} & \rho_3 \bullet \mathsf{join}(I_{sp} \cup \{m_o\}, j_2)
\end{array}
$$

Replacing a sub-pattern involves removing the old structure followed by the overlap of the new pattern. A key operation is $remNodes$ which computes the nodes to be removed.

Definition 10 (The $\mathsf{replaceP}$ pattern). *Let $\rho, \rho_r \in \mathcal{P}$ and X be the set of triples of nodes, where the first component is a node of ρ, the second one is a node of ρ_r and the third is a fresh node in both coordination patterns. Then,*

$$
\rho \bullet \mathsf{replaceP}(\rho_r, X) = (\rho \bullet \mathsf{removeP}(\pi_1(remNodes(\rho, 2^{\pi_1}(X)))) \bullet \mathsf{overlapP}(\rho_r, X))
$$

Finally, the $\mathsf{implodeP}$ pattern collapses a set of nodes taken as the interface of a sub-structure. Formally,

Definition 11 (The $\mathsf{implodeP}$ pattern). *Let $\rho \in \mathcal{P}$, j be a fresh node in ρ and $X \subseteq \mathcal{N}ode$ be a set of nodes of ρ, which represent the border of the structure to implode. Then,*

$$
\begin{array}{rl}
\rho \bullet \mathsf{implodeP}(X, j) = \text{let} & (Ch, N) = remNodes(\rho, X) \\
 & \rho_1 = \rho \bullet \mathsf{removeP}(Ch) \\
 & M = updateNodes(N, \rho_1) \\
\text{in} & \rho_1 \bullet \mathsf{join}(M, j)
\end{array}
$$

where $updateNodes(N, \rho) = N \cap \{x | x \neq \perp \wedge (x = \pi_1(e) \vee x = \pi_4(e)) \wedge e \in \rho\}$

Operation $remNodes$, used above, is defined as follows:

$$
\begin{array}{l}
remNodes : \mathcal{P} \times 2^{\mathcal{N}ode} \to 2^{\mathcal{N}ame} \times 2^{\mathcal{N}ode} \\
remNodes\ (\rho, N) = \\
\quad \text{let} \quad N_1 = N \cup \{x \notin N | (\langle x, id, t, \perp \rangle, \langle a, id, t, \perp \rangle \in \rho) \\
\qquad\qquad\qquad\qquad\qquad \vee (\langle \perp, id, t, x \rangle, \langle \perp, id, t, a \rangle \in \rho) \wedge a \in N\} \\
\qquad\quad N_2 = \bigcup_{n \in N_1} \pi_1(collNodes(n, \emptyset, \{n\}, N_1 \setminus \{n\}) \\
\qquad\quad M_{ids} = \bigsqcup_{n \in N_2} ms(\{\pi_2(e) | e \in \rho \wedge (\pi_1(e) = n \vee \pi_4(e) = n)\}) \\
\quad \text{in} \quad (filter(2, M_{ids}), N_2)
\end{array}
$$

Function $collNodes$ collects all nodes between a given starting one and a set of possible terminal nodes. It aims at identifying the corresponding subgraph.

The arguments are a coordination pattern, a starting node, the nodes in path (an empty set at the beginning), the nodes visited and the terminal nodes:

$$\begin{array}{l}
collNodes : \mathcal{P} \times \mathcal{N}ode \times 2^{\mathcal{N}ode} \times 2^{\mathcal{N}ode} \times 2^{\mathcal{N}ode} \rightarrow 2^{\mathcal{N}ode} \times 2^{\mathcal{N}ode} \\
collNodes(\rho, n, a, v, d) = \\
\text{let } A = \{x | x \neq \bot \wedge (n, \_, \_, x) \in \rho \vee (x, \_, \_, n) \in \rho\} \\
\qquad (acc, vis) = rC(\rho, n, A, v, d) \\
\text{in } \ (a \cup acc, vis)
\end{array}$$

where

$$\begin{array}{llll}
\multicolumn{4}{l}{rC : \mathcal{P} \times \mathcal{N}ode \times 2^{\mathcal{N}ode} \times 2^{\mathcal{N}ode} \times 2^{\mathcal{N}ode} \rightarrow 2^{\mathcal{N}ode} \times 2^{\mathcal{N}ode}} \\
\multicolumn{4}{l}{rC(\rho, n, \emptyset, v, d) = (\emptyset, v)} \\
\multicolumn{4}{l}{rC(\rho, n, (x : xs), v, d) =} \\
\text{if } x \in d & \text{then let} & (a, v_1) = rC(\rho, n, xs, v \cup \{x\}, d) & \\
 & \text{in} & (\{n\} \cup a, v \cup v_1) & \\
\text{else if } x \in v & \text{then} & rC(\rho, n, xs, v \cup \{x\}, d) & \\
\quad \text{else} & \text{let} & (r, v_2) = collNodes(\rho, x, \emptyset, v \cup \{x\}, d) & \\
 & \text{in} & \text{if } r \neq \emptyset \text{ then } (r \cup \{n\}, v_2) & \\
 & & \text{else } rC(\rho, n, xs, v_2, d) &
\end{array}$$

Note that these definitions resort to multi-sets, defined, as usual, as functions from the type of interest to the natural numbers (which encode multiplicities). Typical operations on multisets include *domain* given by $dom(C) = \{a \in A | C(a) \neq 0\}$ and *union*, $(C_1 \sqcup C_2)(a) = C_1(a) + C_2(a)$. Conversion to sets, and back, are also recorded here as $ms : 2^A \rightarrow \mathbb{N}^A$, $ms(A) = [a \rightarrow 1 | a \in A]$, and $filter : \mathbb{N} \times \mathbb{N}^A \rightarrow 2^A$ given by $filter(i, C) = \{x \in dom(C) | C(x) \geq i\}$. The latter converts a multi-set into a set, by filtering the elements that occur at least i times. For a better comprehension of this pattern, the reader may refer to the case study in Section 5.

4 CooPLa: a language for patterns and reconfigurations

Both architectural and reconfiguration patterns can be designed with the help of a domain specific language — CooPLa— and an integrated editor, supplied as a plug-in for Eclipse. It supports syntax colouring and intelligent code-completion and offers during-edition syntax and semantic error checking and error marking for consistent development of patterns. While editing, the tool offers a visualisation of its graph representation, and any change in the code is automatically reflected in this view. Fig. 4 shows a snapshot.

With CooPLa we define communication channels, coordination pattern and reconfigurations.

Channels. Fig. 3 depicts the definition of some of the Reo-like channels introduced above. Note that the lossy channel type extends that of sync (*cf.*, the `extends` keyword). This means the information flow from a to b defined in the latter still applies; only additional behaviour is specified: if there is a request on a but not on b, data will flow through a and lost (*cf.*, `NULL` keyword). Notice the use of $!b$ to explicitly express the absence of requests on b. As another example, consider the drain channel. It has two input ports through which data flows to be lost. The '|' construct means that both flows are performed in parallel. Finally, the FIFO channel has an internal state of type *buffer* specified as a sequence of dimension N and observers E and F on which result depends the channel behaviour.

```
channel sync(a:b){
   a,b -> flow a to b;
}


channel lossy(a:b) extends sync{
   a,!b -> flow a to NULL;
}


channel drain(a,b:) {
   a,b -> flow a to NULL | flow b to NULL;
}
```

```
channel fifo~N(a:b){
   state: buffer;
   observers: E, F;

   // buffer = ELEM*
   // E = buffer.len = 0;
   // F = buffer.len = N;

   a,!F -> flow a to buffer;
   !E,b -> flow buffer to b;
}
```

Fig. 3. The sync, lossy, drain and fifo channels in CooPLa.

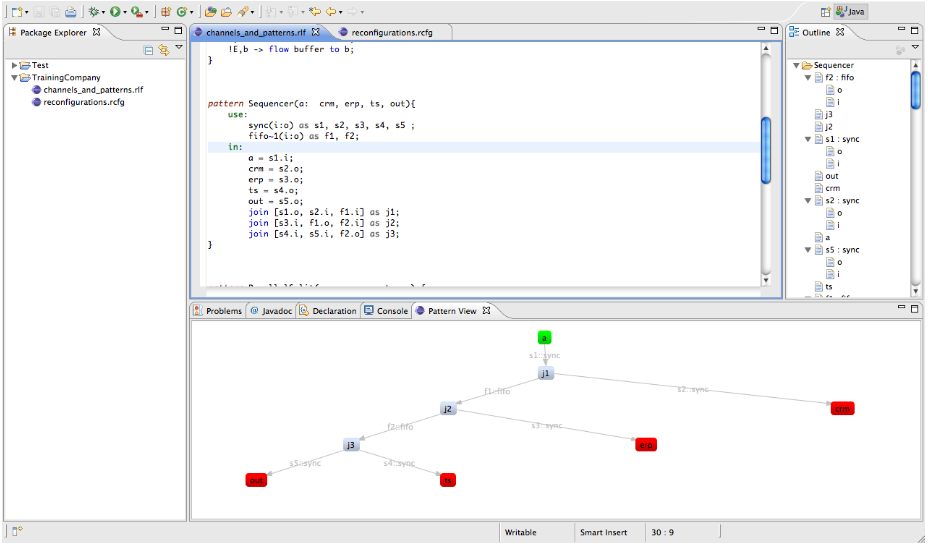
Coordination patterns. Coordination patterns are defined by composition of primitive channels and patterns previously defined. Declaration of instances is preceded by the reserved word use. Each instance is declared by indicating (i) the entity name with the ports locally renamed and (ii) a list of aliases (similar to variables in traditional programming languages) to be used in the subsequent parts of the pattern body definition. In case of instantiating a channel with time or structure, it is defined the inherent dimensions, and in some cases, how such structure is initialised (making use of the observers defined for such structure).

Patterns are composed by interconnecting ports declared in their interfaces. This is achieved by the set of primitive reconfigurations introduced in Definition 2. Fig. 4 shows an example of the Sequencer coordination pattern expressed in the context of the tool developed to support CooPLa.

Reconfigurations. Reconfigurations in CooPLa are also specified compositionally from the primitives given in Definition 2, or from more complex reconfigurations previously defined. Operators over standard data types (*e.g.*, List, Pair and Triple) can also be used: such is the case, in Fig. 5 of the forall structure which iterates over all elements of a list. Application of a reconfiguration r to a pattern ρ is denoted by ρ @ r. Fig. 5 shows an example of two reconfiguration specifications and respective application to instances of coordination patterns. Both Fig. 4 and 5 present parts of the case study addressed next.

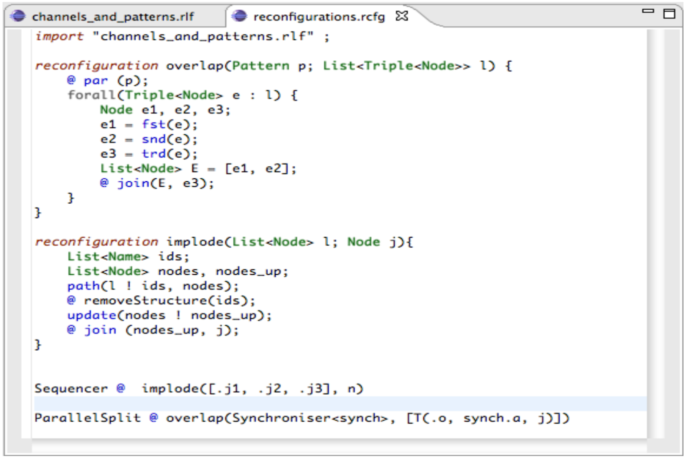
5 Example: A fragment of a case-study

This section illustrates the use of architectural and reconfiguration patterns in a typical example of web-service orchestration for system integration. The case-study from where this example was borrowed involved a professional training company with facilities in six different locations, which relied on four main



```
    !E,b -> flow buffer to b;
}



pattern Sequencer(a:  crm, erp, ts, out){
    use:
        sync(i:o) as s1, s2, s3, s4, s5 ;
        fifo~1(i:o) as f1, f2;
    in:
        a = s1.i;
        crm = s2.o;
        erp = s3.o;
        ts = s4.o;
        out = s5.o;
        join [s1.o, s2.i, f1.i] as j1;
        join [s3.i, f1.o, f2.i] as j2;
        join [s4.i, s5.i, f2.o] as j3;
}
```

Fig. 4. Tool Support for CooPLa



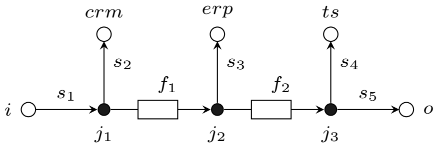
```
import "channels_and_patterns.rlf" ;

reconfiguration overlap(Pattern p; List<Triple<Node>> l) {
    @ par (p);
    forall(Triple<Node> e : l) {
        Node e1, e2, e3;
        e1 = fst(e);
        e2 = snd(e);
        e3 = trd(e);
        List<Node> E = [e1, e2];
        @ join(E, e3);
    }
}

reconfiguration implode(List<Node> l; Node j){
    List<Name> ids;
    List<Node> nodes, nodes_up;
    path(l ! ids, nodes);
    @ removeStructure(ids);
    update(nodes ! nodes_up);
    @ join (nodes_up, j);
}


Sequencer @  implode([.j1, .j2, .j3], n)

ParallelSplit @ overlap(Synchroniser<synch>, [T(.o, synch.a, j)])
```

Fig. 5. Reconfigurations in CooPLa

software components (all working in complete isolation): an *Enterprise Resource Planner* (ERP), a *Customer Relationship Management* (CRM), a *Training Server* (TS), and a *Document Management System* (DMS). The expansion of this company entailed the need for better integration of the whole system. This lead to changing components into services and adopting a SOA solution.

Several problems, however, were found during service orchestration analysis. A recurrent one was the lack of parallelism in the business workflow, slowing the whole system down. The user's information update activity which involves the user update services provided by ERP, CRM and TS components, was one of the tasks affected by such lack of parallel computation, as these services were invoked in sequence.

Let ρ, in Fig. 6, be the coordination pattern (known as a *Sequencer*) used for sequential service orchestration[1]. Resorting to the reconfiguration patterns introduced in Section 3.2, let us rearrange the coordination policy so that user profiles (in each component) are updated in parallel. A possible solution is obtained by applying the implodeP reconfiguration pattern as $\rho \bullet \mathsf{implodeP}(\{j_1, j_3\}, n)$. The following paragraphs show, step-by-step, how to compute the resulting coordination pattern, depicted in Fig. 7. The actual CooPLa script for this reconfiguration is depicted in Fig. 5.



$$\rho = \left\langle \{i\}, \{crm, erp, ts, o\}, \left\{ \begin{array}{c} (i, s_1, \mathsf{sync}, j_1), \quad (j_1, s_2, \mathsf{sync}, crm), \ (j_1, f_1, \mathsf{fifo_e}, j_2), \\ (j_2, s_3, \mathsf{sync}, erp), \quad (j_2, f_2, \mathsf{fifo_e}, j_3), \quad (j_3, s_4, \mathsf{sync}, ts), \\ (j_2, s_5, \mathsf{sync}, o) \end{array} \right\} \right\rangle$$

Fig. 6. The *Sequencer* Coordination Pattern

The first argument of implodeP provides the border nodes of the structure one desires to superpose onto the node in the second argument. From Definition 11 we first identify the complete structure to remove, which is composed of the unique names of the channels and their nodes (the border nodes plus some intermediary ones). Operation *remNodes*, with ρ and $\{j_1, j_3\}$ as arguments, starts by identifying the intermediary nodes and channels to retrieve the relevant structure:

$$\begin{aligned} N_1 &= \{j_1, j_2\} \cup \emptyset = \{j_1, j_2\} \\ N_2 &= \{j_1, j_2\} \cup \{j_3, j_2\} = \{j_1, j_3, j_2\} \\ M_{ids} &= ms(\{s_1, s_2, f_1\}) \sqcup ms(\{f_2, s_4, s_3\}) \sqcup ms(\{f_1, s_3, f_2\}) \\ &= [s_1 \mapsto 1, s_2 \mapsto 1, f_1 \mapsto 2, f_2 \mapsto 2, s_4 \mapsto 1, s_3 \mapsto 1, s_5 \mapsto 1] \\ Ch &= filter(2, M_{ids}) = \{f_1, f_2\} \end{aligned}$$

---

[1] For illustration purposes, the input and output ports of the coordination patterns are shown in the concretization of their formal model

Once the intermediary nodes and channels are identified, we proceed by applying the $\mathsf{removeP}$ reconfiguration pattern as $\rho \bullet \mathsf{removeP}(Ch)$. This boils down to the recursive application of remove: $(\rho \bullet \mathsf{remove}(f_1)) \bullet \mathsf{remove}(f_2)$. Let ρ' be the result of removing f_1 from ρ:

$$\rho' = \left\langle \{i, j_2\}, \{crm, erp, ts, o\}, \left\{ \begin{array}{cc} (i, s_1, \mathsf{sync}, j_1), & (j_1, s_2, \mathsf{sync}, crm), \\ (j_2, s_3, \mathsf{sync}, erp), & (j_2, f_2, \mathsf{fifo_e}, j_3), \\ (j_3, s_4, \mathsf{sync}, ts), & (j_2, s_5, \mathsf{sync}, o) \end{array} \right\} \right\rangle$$

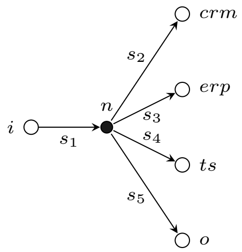
and ρ'' be the result of removing f_2 from ρ', which is actually the outcome of applying the $\mathsf{removeP}$ reconfiguration pattern to ρ:

$$\rho'' = \left\langle \{i, j_2, j_3\}, \{crm, erp, ts, o\}, \left\{ \begin{array}{ll} (i, s_1, \mathsf{sync}, j_1), & (j_1, s_2, \mathsf{sync}, crm), \\ (j_2, s_3, \mathsf{sync}, erp), & (j_3, s_4, \mathsf{sync}, ts), \\ (j_2, s_5, \mathsf{sync}, o) & \end{array} \right\} \right\rangle$$

After removing the channels, set N_2 is updated, to delete nodes that have been removed from the initial coordination pattern, ρ. In this case, N_2 remains unchanged, then:

$$M = \{j_1, j_2, j_3\} \cap \{i, j_1, crm, j_2, erp, j_3, ts, j_3, o\} = \{j_1, j_2, j_3\}$$

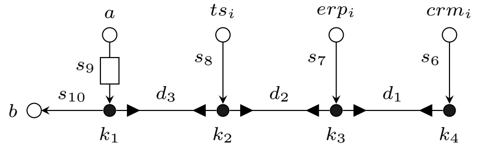
Finally, we merge the nodes of M with node n, and obtain the desired coordination pattern (Fig. 7) which encodes a parallel workflow policy and consequently allows for the update of user's information in parallel.



$$\rho_1 = \left\langle \{i\}, \{crm, erp, ts, o\}, \left\{ \begin{array}{lll} (i, s_1, \mathsf{sync}, n), & (n, s_2, \mathsf{sync}, crm), & (n, s_3, \mathsf{sync}, erp) \\ (n, s_4, \mathsf{sync}, ts), & (n, s_5, \mathsf{sync}, o), & \end{array} \right\} \right\rangle$$

Fig. 7. After *imploding* the Sequencer: the Parallel Split coordination pattern

The resulting pattern actually does the job: the three user update services are called simultaneously, and the flow continues to the output port o, which enables contiguous activities. However, it does not cope with another requirement enforcing that no other activity should start before the user's information is updated. The obvious solution is to delay the flow on port o, until the three services provide a *finish* acknowledgement. A new reconfiguration is, therefore, necessary: we proceed by *overlapping* a *Synchroniser* pattern ρ_s (see Fig. 8). The CooPLa specification of this reconfiguration is shown in Fig. 5. The idea is to connect nodes o and a in such a way that all other input ports of ρ_s are free to connect to the feedback service interface of the CRM, ERP and TS components. Fig. 9



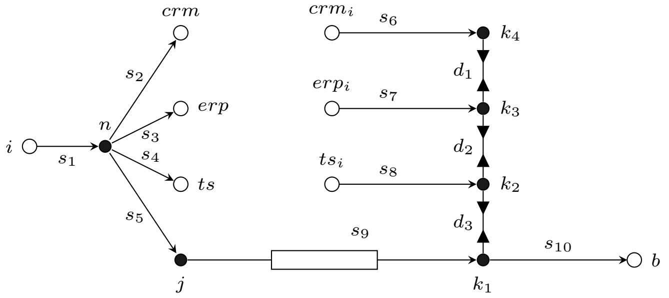
$$\rho_s = \left\langle \{a, crm_i, erp_i, ts_i\}, \{b\}, \left\{ \begin{array}{c} (a, s_9, \mathsf{fifo_e}, k_1), \quad (k_1, s_{10}, \mathsf{sync}, b), \quad (k_1, d_3, \mathsf{drain}, \perp), \\ (k_2, d_3, \mathsf{drain}, \perp), \ (ts_i, s_8, \mathsf{sync}, k_2), \ (k_2, d_2, \mathsf{drain}, \perp), \\ (k_3, d_2, \mathsf{drain}, \perp) \ (erp_i, s_7, \mathsf{sync}, k_3), \ (k_3, d_1, \mathsf{drain}, \perp), \\ (k_4, d_1, \mathsf{drain}, \perp), \ (crm_i, s_8, \mathsf{sync}, k_4) \end{array} \right\} \right\rangle$$

Fig. 8. The *Synchroniser* Coordination Pattern

depicts the result of performing $\rho_1 \bullet \mathsf{overlapP}(\rho_s, \{(o, a, j)\})$. From Definition 8, we start by computing $\rho_1 \bullet \mathsf{par}(\rho_s)$, which yields the following pattern:

$$\rho''' = \left\langle \begin{array}{c} \{i, a, ts_i, erp_i, crm_i\}, \{o, crm, erp, ts, b\}, \\ \left\{ \begin{array}{cccc} (i, s_1, \mathsf{sync}, n), & (n, s_2, \mathsf{sync}, crm), & (n, s_3, \mathsf{sync}, erp) & (n, s_4, \mathsf{sync}, ts), \\ (n, s_5, \mathsf{sync}, o), & (a, s_9, \mathsf{fifo_e}, k_1), & (k_1, s_{10}, \mathsf{sync}, b), & (k_1, d_3, \mathsf{drain}, \perp), \\ (k_2, d_3, \mathsf{drain}, \perp), & (ts_i, s_8, \mathsf{sync}, k_2), & (k_2, d_2, \mathsf{drain}, \perp), & (k_3, d_2, \mathsf{drain}, \perp) \\ (erp_i, s_7, \mathsf{sync}, k_3), & (k_3, d_1, \mathsf{drain}, \perp), & (k_4, d_1, \mathsf{drain}, \perp), & (crm_i, s_8, \mathsf{sync}, k_4) \end{array} \right\} \end{array} \right\rangle$$

Finally, we merge nodes o and a together into a node j, by performing $\rho''' \bullet \mathsf{join}(\{o, a\}, j)$. The result is presented in Fig. 9, which is actually the coordination pattern meeting the requirement of not allowing other activities to start before the user's information update in the CRM, ERP and TS components is completed.



$$\rho_2 = \left\langle \begin{array}{c} \{i, ts_i, erp_i, crm_i\}, \{crm, erp, ts, b\}, \\ \left\{ \begin{array}{cccc} (i, s_1, \mathsf{sync}, n), & (n, s_2, \mathsf{sync}, crm), & (n, s_3, \mathsf{sync}, erp) & (n, s_4, \mathsf{sync}, ts), \\ (n, s_5, \mathsf{sync}, j), & (j, s_9, \mathsf{fifo_e}, k_1), & (k_1, s_{10}, \mathsf{sync}, b), & (k_1, d_3, \mathsf{drain}, \perp), \\ (k_2, d_3, \mathsf{drain}, \perp), & (ts_i, s_8, \mathsf{sync}, k_2), & (k_2, d_2, \mathsf{drain}, \perp), & (k_3, d_2, \mathsf{drain}, \perp) \\ (erp_i, s_7, \mathsf{sync}, k_3), & (k_3, d_1, \mathsf{drain}, \perp), & (k_4, d_1, \mathsf{drain}, \perp), & (crm_i, s_8, \mathsf{sync}, k_4) \end{array} \right\} \end{array} \right\rangle$$

Fig. 9. Overlapping Parallel Split with the Synchroniser pattern

6 Conclusions

6.1 Related work

Reconfigurations in SOA are, most of the times, focused on replacing services, or modifying their connections to the coordination layer. Often they neglect structural changes in the actual interaction layer itself [13,17]. In [19,18], however, the authors highlight the role played by software connectors during runtime architectural changes. Although these changes are again focused on the manipulation of components, they recognise that connectors are also amenable to contextual adaptations in order to keep the consistency of the architecture.

Reference [24] resorts to category theory to model software architectures as labelled graphs of components and connectors. Reconfigurations are modelled via algebraic graph rewriting rules. This approach has some points of contact with our strategy.

In [5], are presented two approaches for modelling architectures, their styles, dynamism and properties based on pioneer work on graph rewriting [9]. The first approach [7,4] represent the architecture as a hypergraph with hyperedges and ports to address components and connectors. The Alloy framework is used to specify the graph of the architecture as well as structural and behavioural properties of the architecture dynamism. Reconfigurations are performed via graph rewriting productions and graph morphisms. The second approach [5,6] adopts a hierarchical model of the graph, to which they refer to as designs, where the architecture constituents are represented uniformly and interfaces are added to the graph, allowing for reuse. The Maude framework is used to specify the architecture and to perform analysis on structural and behavioural properties. Reconfigurations are encoded as rewrite rules over terms.

In [16,15], the authors relay on high-level replacement systems, more precisely on typed hypergraphs, to describe Reo connectors (and architectures, in general). In this perspective, vertices are the nodes and (typed hyper-) edges are communication channels and components. Reconfiguration rules are specified as graph productions for pattern matching. This approach performs atomic complex reconfigurations, rather than a sequence of basic modifications, which is stated as an advantage for maintaining system consistency. Nevertheless, the model may become too complex even when a simple primitive operation needs to be applied.

Differently, in [8] architectures are modelled as Reo connectors, and no information on components is stored in the model. The model is a triple composed of channels with a type and distinct named ports, a set of visible nodes and a set of hidden nodes. Their model is similar to ours, but for the distinction introduced here between input and output nodes and the need we avoid to be explicit on the hidden nodes of a pattern. Although a number of primitive transformations are proposed, this work, as most of the others, do not consider ‘big-step’ reconfigurations which seems a severe limitation in practice.

6.2 Summary and future work

The paper introduces a model for reconfiguration of coordination patterns, described as a graphs of primitive channels. It is shown how typical reconfiguration patterns can be expressed in the model by composition of elementary transformations. CooPLa provides a setting to animate and experiment reconfigurations upon typical coordination patterns. We are currently involved in their classification in a suitable ontology, taking into account structural and behavioural properties of coordination patterns. What is still missing, however, is the inclusion in the model of automatic assessing mechanisms to assess reconfigurations semantically and trigger their application based on non-functional requirements. This concern is orthogonal to the work presented in this paper.

References

1. Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services: Concepts, Architectures and Applications.* Springer Publishing Company, Incorporated, 1st edition, 2010.
2. Farhad Arbab. Reo: a channel-based coordination model for component composition. *Mathematical. Structures in Comp. Sci.*, 14(3):329–366, June 2004.
3. Farhad Arbab and Farhad Mavaddat. Coordination through channel composition. In Farhad Arbab and Carolyn Talcott, editors, *Coordination Models and Languages*, volume 2315 of *Lecture Notes in Computer Science*, chapter 6, pages 275–297. Springer Berlin / Heidelberg, Berlin, Heidelberg, March 2002.
4. M. Beek, A. Bucchiarone, and S. Gnesi. Dynamic software architecture development: Towards an automated process. In *Software Engineering and Advanced Applications, 2009. SEAA 2009. 35th Euromicro Conference on*, pages 105–108. IEEE, August 2009.
5. Roberto Bruni, Antonio Bucchiarone, Stefania Gnesi, Dan Hirsch, and Alberto Lluch Lafuente. Graph-Based design and analysis of dynamic software architectures concurrency, graphs and models. In Pierpaolo Degano, Rocco Nicola, and José Meseguer, editors, *Concurrency, Graphs and Models*, volume 5065 of *Lecture Notes in Computer Science*, chapter 4, pages 37–56. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2008.
6. Roberto Bruni, Howard Foster, Alberto L. Lafuente, Ugo Montanari, and Emilio Tuosto. A formal support to business and architectural design for service oriented systems. In Martin Wirsing and Matthias Hölzl, editors, *Rigorous software engineering for service-oriented systems*, chapter A formal support to business and architectural design for service-oriented systems, pages 133–152. Springer-Verlag, Berlin, Heidelberg, 2011.
7. Antonio Bucchiarone. *Dynamic Software Architectures for Global Computing Systems.* PhD thesis, IMT Institute for Advanced Studies, Lucca, Lucca, Italy, 2008.
8. Dave Clarke. A basic logic for reasoning about connector reconfiguration. *Fundam. Inf.*, 82:361–390, February 2008.
9. Pierpaolo Degano and Ugo Montanari. A model for distributed systems based on graph rewriting. *J. ACM*, 34(2):411–449, April 1987.
10. Thomas Erl. *SOA Design Patterns.* Prentice Hall PTR, 1 edition, January 2009.

11. José L. Fiadeiro. Software services: Scientific challenge or industrial hype? In Zhiming Liu and Keijiro Araki, editors, *Theoretical Aspects of Computing - ICTAC 2004*, volume 3407 of *Lecture Notes in Computer Science*, pages 1–13. Springer Berlin / Heidelberg, 2005.
12. H. Gomaa and M. Hussein. Software reconfiguration patterns for dynamic evolution of software architectures. In *Software Architecture, 2004. WICSA 2004. Proceedings. Fourth Working IEEE/IFIP Conference on*, pages 79–88, 2004.
13. Petr Hnětynka and František Plášil. Dynamic reconfiguration and access to services in hierarchical component models. In Ian Gorton, George T. Heineman, Ivica Crnković, Heinz W. Schmidt, Judith A. Stafford, Clemens Szyperski, and Kurt Wallnau, editors, *Component-Based Software Engineering*, volume 4063 of *Lecture Notes in Computer Science*, chapter 27, pages 352–359. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2006.
14. J. Kramer and J. Magee. Dynamic configuration for distributed systems. *Software Engineering, IEEE Transactions on*, SE-11(4):424–436, 1985.
15. C. Krause, Z. Maraikar, A. Lazovik, and F. Arbab. Modeling dynamic reconfigurations in Reo using high-level replacement systems. *Science of Computer Programming*, 76(1):23–36, 2011.
16. Christian Krause. *Reconfigurable Component Connectors.* PhD thesis, Leiden University, Amsterdam, The Netherlands, 2011.
17. M. Malohlava and T. Bures. Language for reconfiguring runtime infrastructure of component-based systems. In *Proceedings of MEMICS 2008*, Znojmo, Czech Republic, November 2008.
18. Nenad Medvidovic. ADLs and dynamic architecture changes. In *Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops*, ISAW '96, pages 24–27, New York, NY, USA, 1996. ACM.
19. Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-based runtime software evolution. In *Proceedings of the 20th international conference on Software engineering*, ICSE '98, pages 177–186, Washington, DC, USA, 1998. IEEE Computer Society.
20. George A. Papadopoulos and Farhad Arbab. Coordination models and languages. *Advances in Computers*, 46:330–401, 1998.
21. Andres J. Ramirez and Betty H. C. Cheng. Design patterns for developing dynamically adaptive systems. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '10, pages 49–58, New York, NY, USA, 2010. ACM.
22. Nick Russell, Arthur H. M. ter Hofstede, Wil M. P. van der Aalst, and Natalya Mulyar. Workflow Control-Flow patterns: A revised view. Technical report, BPM-center.org, 2006.
23. Lionel Seinturier, Philippe Merle, Romain Rouvoy, Daniel Romero, Valerio Schiavoni, and Jean-Bernard Stefani. A Component-Based Middleware Platform for Reconfigurable Service-Oriented Architectures. *Software: Practice and Experience*, 2011.
24. Michel Wermelinger and José L. Fiadeiro. Algebraic software architecture reconfiguration. In *Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-7, pages 393–409, London, UK, 1999. Springer-Verlag.

25. Uwe Zdun, Carsten Hentrich, and Wil M. P. Van Der Aalst. A survey of patterns for Service-Oriented architectures. *Int. J. Internet Protoc. Technol.*, 1:132–143, May 2006.