Smooth Orchestrators*

Cosimo Laneve[1] and Luca Padovani[2]

[1] Department of Computer Science, University of Bologna
laneve@cs.unibo.it
[2] Information Science and Technology Institute, University of Urbino
padovani@sti.uniurb.it

Abstract. A *smooth orchestrator* is a process with several alternative branches, every one defining synchronizations among co-located channels. Smooth orchestrators constitute a basic mechanism that may express standard workflow patterns in Web services as well as common synchronization constructs in programming languages. Smooth orchestrators may be created in one location and migrated to a different one, still not manifesting problems that usually afflict generic mobile agents.

We encode an extension of Milner's (asynchronous) pi calculus with join patterns into a calculus of smooth orchestrators and we yield a strong correctness result (full abstraction) when the subjects of the join patterns are co-located. We also study the translation of smooth orchestrators into finite-state automata, therefore addressing the implementation of co-location constraints and the case when synchronizations are not linear with respect to subjects.

1 Introduction

Web services programming languages use mechanisms for defining services available on the Web. Examples of these languages are Microsoft XLANG [13] and its visual environment BizTalk, IBM WSFL [10], BPEL [2], WS-CDL [8], and WSCI [8]. Among the basic mechanisms used by such technologies, there are the so-called *orchestrators*, which compose available services, possibly located at different administrative domains, by adding a central coordinator that is responsible for invoking and combining sub-activities.

This contribution addresses a very simple class of orchestrators, those triggering a continuation when a pattern of messages on a set of services is available. For example, the orchestrator

$$x(u) \,\&\, y(v) \triangleright \overline{z}\, uv$$

enables the continuation $\overline{z}\, uv$ if one message to the service x and one message to the service y are available. The orchestrator expires once it has executed. This process is easy to implement if the two services x and y – called *channels* in the following – are co-located: it suffices to migrate $x(u) \,\&\, y(v) \triangleright \overline{z}\, uv$ to the location of x and y. The general case when the continuation $\overline{z}\, uv$ is a large process may be always reduced to the simpler one.

* Aspects of this investigation were partly supported by a Microsoft initiative in concurrent computing and Web services.

L. Aceto and A. Ingólfsdóttir (Eds.): FOSSACS 2006, LNCS 3921, pp. 32–46, 2006.

If x and y are not co-located then we immediately face a *global consensus problem*: the location running the channel(-manager)s for x and y must agree with the one running $x(u) \,\&\, y(v) \triangleright \overline{z}\, uv$ for consuming outputs. (Migrating $x(u) \,\&\, y(v) \triangleright P$, or a variant of it, to the location of x or of y does not simplify the problem because x and y are not co-located.) Observe that similar problems are manifested by orchestrators such as

$$x(u) \triangleright \overline{z}\, u \quad + \quad y(v) \triangleright \overline{z'}\, v$$

where "+" picks either one of $x(u) \triangleright \overline{z}\, u$ or $y(v) \triangleright \overline{z'}\, v$ according to the availability of a message on x or on y.

A language with orchestrators should therefore simply disallow those ones that combine not co-located channels, on the grounds that they are un-implementable. There are several ways for removing such problematic orchestrators. The join calculus [6] achieves (co-)locality with an elegant syntactic constraint in which the same language construct is used both to declare channels and to define their continuations. Channels that are being orchestrated are, by definition, co-located. For example, the process

$$\begin{aligned} x(u) \triangleright (y, z)(\ & \overline{s}\, uy \mid \overline{t}\, uz \\ & \mid y(v) \triangleright P + z(w) \triangleright Q) \end{aligned}$$

specifies a service x that starts the sub-activities s and t with local channels y and z, respectively. The orchestrator – defined on these local channels – takes into account the first activity that completes. It is worth to remark that the above process remains implementable even if y and z are not co-located with x (this may be the case when, for load-balancing reasons, it is preferable to create the two channels remotely). To formalize this constraint on y and z it suffices to add an explicit co-location construct to channel definitions. Write $z \mathbin{@} y$ to mean that z is created at the same location as y. Then the above process may be rewritten as $x(u) \triangleright (y \mathbin{@} y, z \mathbin{@} y)(\overline{s}\, uy \mid \overline{t}\, uz \mid y(v) \triangleright P + z(w) \triangleright Q)$, where the constraint $y \mathbin{@} y$ means that y may be created at whatever location.

We also consider a further relaxation of the join calculus locality constraint, which combines co-location and input capability. Input capability is the ability to receive a channel name and subsequently accept inputs on it. Orchestrators that join received channels and locally defined ones are again implementable if all the channels are co-located. In this case, the co-location constraint may be enforced statically if the language has mechanisms for extracting the location information out of received channels. In practice this is trivial because channels contains the IP addresses of their location. Technically we write $x(u \mathbin{@} u) \,\&\, y(v \mathbin{@} u) \triangleright P$ to select messages on x and y that carry co-located channels. In this case, the continuation P could orchestrate u and v, that is P might be $u(u' \mathbin{@} u') \,\&\, v(v' \mathbin{@} v') \triangleright P'$.

We end up in considering a class of orchestrators, which we call *smooth* (the terminology is drawn from [1]), that consist of several alternative branches, every one defining synchronizations among co-located channels and having an output as continuation. The implementation of (smooth) orchestrators poses a number of challenges because they may be dynamically created and because of the co-location constraints that may introduce dependencies between different channels in the same join pattern. With respect to Le Fessant and Maranget's compilation technique of join patterns [9]

we discuss a number of extensions for implementing smooth orchestrators of increasing complexity. In particular, we show that it is still possible to use finite state automata for handling join pattern definitions, even when the joined channels are not fresh.

Related work. Distributed implementations of input-guarded choices have already been studied in detail by the pi community. Nestmann and Pierce have proposed the following encoding of the orchestrator $\sum_{i\in 1..n} x_i(u_i)\triangleright P$ (we rewrite the solution in our notation):

$$(\ell \mathbin{@} \ell)(\overline{\ell}\, t \mid \prod_{i\in 1..n} x_i(u_i \mathbin{@} u_i) \triangleright \Big(\ell(u \mathbin{@} t) \triangleright (\overline{\ell}\, f \mid P) \quad + \quad \ell(u \mathbin{@} f) \triangleright (\overline{\ell}\, f \mid \overline{x_i}\, u_i)\Big)$$

where t and f are two free channels that are not co-located. While this technique may be refined so that every branch of the choice inputs on different channels ℓ (*c.f.* linear forwarders [7]), it seems not useful for our orchestrators where guards are complex input patterns. Implementations of "defined once"-orchestrators on co-located channels have been studied in detail for the join calculus in [9]. There are similarities between our calculus and MAGNETs [3]. In MAGNETs orchestrators are implemented as agents that migrate to the location where the synchronized channels are defined. As in this paper, MAGNETs only synchronize co-located channels, even though this condition is not enforced by a type system.

The calculus of orchestrators that we study is actually intermediate between pi calculus [11] and join calculus [6]. Its motivations are pretty practical. We have recently developed a distributed machine for the pi calculus – `PiDuce` [5, 4] – where it is possible to create channels and their managers in remote locations. This machine supports inputs on received channels by decoupling them into a particle migrating to the remote channel (the linear forwarder) and the continuation. This contribution analyzes an extension of this feature with patterns of inputs and discusses the technical problems we found in prototyping them. Smooth orchestrators that coordinate local channels have been already implemented in the current `PiDuce` prototype [4]. In the next release we expect to extend the prototype with migrating smooth orchestrators and co-location constraints.

Plan of paper. The paper is structured as follows. Section 2 gives the calculus with orchestrators, and its reference semantics – barbed congruence. Section 3 gives the encoding of few sample workflow patterns. Section 4 gives the smoothness constraint on orchestrators that makes them implementable. We demonstrate the invariance of the constraint with respect to the reduction and the implementation of the full calculus. Section 5 describes the implementation of smooth orchestrators.

2 Processes with Orchestrators

In this section we introduce the calculus with orchestrators. We first present the syntax, then the co-location relation, which is preparatory to the operational semantics, and finally the operational semantics.

2.1 Syntax

We assume an infinite set of *names* ranged over by x, u, v, ... Names represent communication channels, which are also the values being transmitted in communications. We write $\widetilde{x}$ for a (possibly empty) finite sequence $x_1 \cdots x_n$ of names. *Name substitutions* $\{\widetilde{y}/\widetilde{x}\}$ are as usual and ranged over ρ, ρ'. We let $\mathtt{dom}(\{\widetilde{y}/\widetilde{x}\}) = \widetilde{x}$. We also write $(x_1, \cdots, x_n \mathbin{@} y_1, \cdots, y_n)$ for $(x_1 \mathbin{@} y_1) \cdots (x_n \mathbin{@} y_n)$. These sequences, called *co-location sequences*, are ranged over by Λ, Λ'.

The syntax consists of *processes* P and *join patterns* J:

P ::=		**processes**	J ::=		**join patterns**
	$\mathsf{0}$	(nil)		$x(\widetilde{u} \mathbin{@} \widetilde{v})$	(input)
$\mid$	$\overline{x}\,\widetilde{u}$	(output)	$\mid$	$J \,\&\, J$	(join)
$\mid$	$\sum_{i \in I} J_i \triangleright P_i$	(orchestrator)			
$\mid$	$(x \mathbin{@} y)P$	(new)			
$\mid$	$P \mid P$	(parallel)			
$\mid$	$!P$	(replication)			

In the rest of the paper, we write $\prod_{i \in 1..n} P_i$ for $P_1 \mid \cdots \mid P_n$ and $J_1 \triangleright P_1 + \cdots + J_n \triangleright P_n$ for $\sum_{i \in 1..n} J_i \triangleright P_i$. We also write $(x)P$ for $(x \mathbin{@} x)P$ and $x(u)$ for $x(u \mathbin{@} u)$.

Free and bound names are standard: x is *bound* in $(x \mathbin{@} y)P$ and $\widetilde{u}$ is bound in $x(\widetilde{u} \mathbin{@} \widetilde{v})$; names are *free* when they occur non-bound. Write $\mathtt{bn}(P)$ and $\mathtt{bn}(J)$ for the bound names of P and J, respectively; similarly write $\mathtt{fn}(P)$ and $\mathtt{fn}(J)$ for the free names. For example, $\mathtt{fn}(x(u \mathbin{@} u) \,\&\, y(v \mathbin{@} u)) = \mathtt{fn}(x(v \mathbin{@} u) \,\&\, y(u \mathbin{@} u)) = \{x, y\}$. The scope of the name x in $(x \mathbin{@} y)P$ is y and the process P; the scope of a name bound by J in $J \triangleright P$ is J and P. The name x in $\overline{x}\,\widetilde{u}$ and in $x(\widetilde{u} \mathbin{@} \widetilde{v})$ is called *subject*; $\mathtt{sn}(J)$ collects all the subjects of J. The names $\bigcup_{i \in I} \widetilde{u}_i$ in $\&_{i \in I} x_i(\widetilde{u}_i \mathbin{@} \widetilde{v}_i)$ are called *defined names*.

Processes define the computational entities of the calculus. Most of the operators are standard from the pi calculus [11], except new $(x \mathbin{@} y)P$, orchestrator $\sum_{i \in I} J_i \triangleright P_i$, and input $x(\widetilde{u} \mathbin{@} \widetilde{v})$. The process $(x \mathbin{@} y)P$ creates a channel x at the same location of y. The process $(x \mathbin{@} x)P$ creates a channel x at a fresh location. The term $x \mathbin{@} y$ in new and input is called *co-location pair*. Orchestrators are reminiscent of join calculus definitions [6] and pi calculus input guarded choices. A branch $J_i \triangleright P_i$ is chosen provided a pattern of outputs that matches with J_i is present. In this case the continuation P_i is run and all the other alternatives are discarded. A pattern of outputs $\overline{x_1}\,\widetilde{u}_1 \mid \cdots \mid \overline{x_n}\,\widetilde{u}_n$ matches with $x_1(\widetilde{v}_1 \mathbin{@} \widetilde{w}_1) \,\&\, \cdots \,\&\, x_n(\widetilde{v}_n \mathbin{@} \widetilde{w}_n)$ provided $\widetilde{u}_i$ and $\widetilde{v}_i$ have the same length and the location constraints in $\widetilde{w}_i$ are satisfied. For example, $\overline{x}\,u \mid \overline{z}\,u'$ matches with $x(v) \,\&\, z(v' \mathbin{@} v)$ if u and u' are two co-located channels.

Join patterns J satisfy the following *well-formedness constraints*:

1. *defined names of J are pairwise different;*
2. *(left-constraining) if $J = \&_{i \in 1..n} x_i(\widetilde{u}_i \mathbin{@} \widetilde{v}_i)$ then $(\widetilde{u}_1 \cdots \widetilde{u}_n \mathbin{@} \widetilde{v}_1 \cdots \widetilde{v}_n)$ is such that, for every decomposition $(\widetilde{u}' \mathbin{@} \widetilde{v}')(u \mathbin{@} v)(\widetilde{u}'' \mathbin{@} \widetilde{v}'')$ of it, we have $v \notin \widetilde{u}''$. With an abuse of terminology, a co-location sequence that satisfies this property is said left-constraining.*

Remark 1. Left-constraining makes & not commutative. For example the pattern $J = x(u \mathbin{@} u) \,\&\, y(v \mathbin{@} u)$ is well-formed while $J' = y(v \mathbin{@} u) \,\&\, x(u \mathbin{@} u)$ is not. Left-constraining makes join patterns parsable from left to right; on the contrary, in $y(v \mathbin{@} u) \,\&\, x(u \mathbin{@} u)$, to find the binder for the occurrence of u in $y(v \mathbin{@} u)$ one has to read the whole join pattern. Left-constraining also simplifies some technical discussions later in the paper.

Remark 2. The well-formedness condition on join patterns does not enforce their linearity with respect to subject names. For example, the pattern $x(u) \,\&\, x(v)$ is well-formed. This linearity constraint is not easy to formalize because, in our calculus, received names may be used as subjects of inputs in the continuations – *input capability*. Removing the feature of input capability, the linearity constraint may be defined as in join calculus [6].

2.2 Co-location Relation

Process reduction is possible if the co-location constraints specified in the join pattern are fulfilled. This fulfillment is defined in terms of the *co-location relation*.

Definition 1. *Let $\widetilde{x} \mathbin{@} \widetilde{y} \vdash u^\frown v$, called the* co-location relation*, be the equivalence relation on names that is induced by the following rules:*

$$\frac{}{(\widetilde{x} \mathbin{@} \widetilde{y})(u \mathbin{@} v) \vdash u^\frown v}\ (\textsc{base}) \qquad \frac{\widetilde{x} \mathbin{@} \widetilde{y} \vdash u^\frown v \quad u, v \neq z}{(\widetilde{x} \mathbin{@} \widetilde{y})(z \mathbin{@} z') \vdash u^\frown v}\ (\textsc{lift})$$

For example $(x \mathbin{@} y)(z \mathbin{@} y) \vdash x^\frown z$ by transitivity. A less evident entailment is $(x \mathbin{@} y)(z \mathbin{@} y)(y \mathbin{@} u) \vdash x^\frown z$. This is due to $(x \mathbin{@} y)(z \mathbin{@} y) \vdash x^\frown z$ and to the (LIFT) rule because $x, z \neq y$. We write $\widetilde{x} \mathbin{@} \widetilde{y} \vdash u_1 \cdots u_n{}^\frown v_1 \cdots v_n$ if $\widetilde{x} \mathbin{@} \widetilde{y} \vdash u_i{}^\frown v_i$ for every i.

The co-location relation induces a partition on names that is left informal in this contribution. For instance $(x \mathbin{@} y)(z \mathbin{@} y)(y \mathbin{@} u)$ gives the partition $\{x, z\}, \{y, u\}$. There are permutations of co-location sequences that preserve the induced partition. A relevant one is the following.

Proposition 1. *If $x \neq x'$, $x \neq y'$, and $x' \neq y$ then $\Lambda(x \mathbin{@} y)(x' \mathbin{@} y') \vdash u^\frown v$ implies $\Lambda(x' \mathbin{@} y')(x \mathbin{@} y) \vdash u^\frown v$.*

Proof. For brevity we only examine four cases, and we assume $u \neq v$.

($u, v \neq x, x'$) From $\Lambda(x \mathbin{@} y)(x' \mathbin{@} y') \vdash u^\frown v$ and the hypotheses we derive $\Lambda \vdash u^\frown v$. From this and the hypotheses we conclude $\Lambda(x' \mathbin{@} y')(x \mathbin{@} y) \vdash u^\frown v$ by (LIFT).

($u = x$, $u, v \neq x'$) From $\Lambda(u \mathbin{@} y)(x' \mathbin{@} y') \vdash u^\frown v$ we derive $\Lambda(u \mathbin{@} y) \vdash u^\frown v$. There are two sub-cases:

- **($v = y$)** We conclude $\Lambda(x' \mathbin{@} y')(u \mathbin{@} v) \vdash u^\frown v$ by (BASE).
- **($v \neq y$)** Since $u^\frown y$ we must have derived $u^\frown v$ by transitivity from $\Lambda \vdash v^\frown y$. From $\Lambda \vdash v^\frown y$ and the hypotheses $v, y \neq x', u$ we get $\Lambda(x' \mathbin{@} y')(u \mathbin{@} y) \vdash v^\frown y$. From this and $\Lambda(x' \mathbin{@} y')(u \mathbin{@} y) \vdash u^\frown y$ by transitivity we obtain $\Lambda(x' \mathbin{@} y')(u \mathbin{@} y) \vdash u^\frown v$.

($u = x'$, $u, v \neq x$) We have $\Lambda(x \mathbin{@} y)(u \mathbin{@} y') \vdash u^\frown v$. There are two sub-cases:

- ($v = y'$) We have $\Lambda(u \mathbin{@} v) \vdash u^\frown v$. From this and the hypotheses $u, v \neq x$ we conclude $\Lambda(u \mathbin{@} v)(x \mathbin{@} y) \vdash u^\frown v$.
- ($v \neq y'$) Since $u^\frown y'$ we must have derived $u^\frown v$ by transitivity from $\Lambda(x \mathbin{@} y) \vdash v^\frown y'$. From this and the hypotheses $v, y' \neq x$ we get $\Lambda \vdash v^\frown y'$. From this and the hypotheses $v, y' \neq u, x$ we get $\Lambda(u \mathbin{@} y')(x \mathbin{@} y) \vdash v^\frown y'$. Using similar arguments we derive $\Lambda(u \mathbin{@} y')(x \mathbin{@} y) \vdash u^\frown y'$ and by transitivity we conclude $\Lambda(u \mathbin{@} y')(x \mathbin{@} y) \vdash u^\frown v$.

($u = x$, $v = x'$) We have $\Lambda(u \mathbin{@} y)(v \mathbin{@} y') \vdash u^\frown v$. From the hypotheses we have $u, v \neq y'$ and $u, v \neq y$. We must have concluded $u^\frown v$ by transitivity from $y^\frown y'$. From this and $y, y' \neq v, u$ we get $\Lambda(v \mathbin{@} y')(u \mathbin{@} y) \vdash y^\frown y'$. From this and $\Lambda(v \mathbin{@} y')(u \mathbin{@} y) \vdash u^\frown v$ we conclude $\Lambda(v \mathbin{@} y')(u \mathbin{@} y) \vdash u^\frown v$. □

It is possible to establish a relation between the partition induced by a co-location sequence and the one obtained when bound names in a suffix of the same sequence are substituted.

Proposition 2. *Let ρ be a substitution such that $\Lambda \vdash \widetilde{x}\rho^\frown\widetilde{y}\rho$. Then $\Lambda(\widetilde{x} \mathbin{@} \widetilde{y}) \vdash u^\frown v$ implies $\Lambda \vdash u\rho^\frown v\rho$.*

Proof. By induction on the proof of $\Lambda(\tilde{x} \mathbin{@} \tilde{y}) \vdash u^\frown v$. The base case is straightforward. The inductive case is when the last rule is an instance of (LIFT). Let $(\widetilde{x} \mathbin{@} \widetilde{y}) = (\widetilde{x}'' \mathbin{@} \widetilde{y}'')(x' \mathbin{@} y')$ and let $\Lambda(\widetilde{x} \mathbin{@} \widetilde{y}) \vdash u^\frown v$ be demonstrated by (LIFT) with premises $\Lambda(\widetilde{x}'' \mathbin{@} \widetilde{y}'') \vdash u^\frown v$ and $u, v \neq x'$. We have that $\Lambda \vdash \widetilde{x}\rho^\frown\widetilde{y}\rho$ implies $\Lambda \vdash \widetilde{x}''\rho^\frown\widetilde{y}''\rho$. Henceforth, by inductive hypothesis, we conclude $\Lambda \vdash u\rho^\frown v\rho$. □

2.3 Operational Semantics

The operational semantic is defined by means of a structural congruence that equates all processes that have essentially the same structure and that are never distinguished.

Definition 2. *Structural congruence $\equiv$ is the smallest equivalence relation that satisfies the following axioms and is closed with respect to contexts and alpha-renaming:*

$$P \mid \mathsf{0} \equiv P \qquad P \mid Q \equiv Q \mid P \qquad P \mid (Q \mid R) \equiv (P \mid Q) \mid R \qquad !P \equiv P \mid !P$$
$$(x \mathbin{@} y)\mathsf{0} \equiv \mathsf{0} \qquad (x \mathbin{@} y)(P \mid Q) \equiv P \mid (x \mathbin{@} y)Q \quad \textit{if } x \notin \mathtt{fn}(P)$$
$$(x \mathbin{@} y)(x' \mathbin{@} y')P \equiv (x' \mathbin{@} y')(x \mathbin{@} y)P \quad \textit{if } x \neq x',\ x \neq y',\ \textit{and } y \neq x'$$

Notice that the last congruence axiom is strictly related to Proposition 1.

Definition 3. *The reduction relation $\rightarrow$ is the least relation satisfying the rule*

$$\frac{M = \prod_{j\in 1..n} \overline{x_j}\,\widetilde{u}_j \quad J_k = \&_{j\in 1..n} x_j(\widetilde{u}_j \mathbin{@} \widetilde{v}_j)^{\,j\in 1..n} \quad k \in I \qquad \mathtt{dom}(\rho) = \bigcup_{j\in I} \widetilde{u}_j \quad \Big(\widetilde{z} \mathbin{@} \widetilde{y} \vdash \widetilde{u}_j\rho^\frown\widetilde{v}_j\rho\Big)}{(\widetilde{z} \mathbin{@} \widetilde{y})\Big(M\rho \mid \sum_{i\in I} J_i \triangleright P_i \mid R\Big) \rightarrow (\widetilde{z} \mathbin{@} \widetilde{y})\Big(P_k\rho \mid R\Big)}$$

and closed under $\equiv$, $\_ \mid \_$ and $(z \mathbin{@} z')\_$.

The last premise of the reduction rule requires that the interacting processes are underneath a sequence of news that is long enough. For example, the process $P = \overline{x}\,u \mid x(u \mathbin{@} v) \triangleright Q$ is inactive. On the contrary $(u \mathbin{@} v)P$ reduces to Q. We refer to the introduction for few sample processes in our calculus.

The semantics is completed by the notion of *barbed congruence* [12]. According to this notion, two processes are considered equivalent if their reductions match and they are indistinguishable under global observations and under any context.

Definition 4. *The name x is a* barb *of P, written $P \downarrow x$, when*

$$\begin{array}{ll} \overline{x}\,\widetilde{u} \downarrow x & \\ (z \mathbin{@} y)P \downarrow x & \text{if } P \downarrow x \text{ and } x \neq z \\ !P \downarrow x & \text{if } P \downarrow x \\ P \mid Q \downarrow x & \text{if } P \downarrow x \text{ or } Q \downarrow x \end{array}$$

Write $\Rightarrow$ for $\rightarrow^*$ and $\Downarrow$ for $\Rightarrow\downarrow$.

A barbed bisimulation *is a symmetric relation ϕ such that whenever $P\ \phi\ Q$ then (1) $P \downarrow x$ implies $Q \Downarrow x$, and (2) $P \rightarrow P'$ implies $Q \Rightarrow Q'$ and $P'\ \phi\ Q'$. The largest barbed bisimulation is noted $\dot{\approx}$.*

Let $C[\,]$ be the set of contexts generated by the grammar:

$$C[\,] \ ::= \quad [\cdot] \quad \mid \quad \sum_{i \in I} J_i \triangleright C[\,] \quad \mid \quad (x \mathbin{@} y)C[\,] \quad \mid \quad P|C[\,] \quad \mid \quad C[\,]|P \quad \mid \quad !C[\,]$$

The barbed congruence *is the largest symmetric relation $\approx$ such that whenever $P \approx Q$ then, for all contexts $C[\,]$, $C[P] \mathrel{\dot{\approx}} C[Q]$.*

3 On the Expressivity of Orchestrators

Orchestrators constitute a basic mechanism that may express standard workflow patterns in Web services as well as common synchronization constructs in programming languages. A few paradigmatic encodings of patterns are described below.

Client/supplier/bank interaction. The first example describes a $Supplier$ that waits for requests from clients. Upon receiving a buy request, the supplier asks the client about his financial availability. The client must reassure the supplier by letting his financial institution (a $bank$) vouch directly for him. In the meantime, the supplier forwards the client's request to the appropriate manufacturer, which will proceed with the delivery as soon as he receives a confirmation from the bank. We write $\overline{x}\,[\widetilde{u}]$ instead of $\overline{x}\,\widetilde{u}$:

$$\begin{array}{rcl} Supplier & \stackrel{\text{def}}{=} & buy(item, x) \triangleright (voucher \mathbin{@} item)(\\ & & \quad \overline{x}\,[voucher, amount] \\ & & \quad \mid voucher(u) \mathbin{\&} item(v) \triangleright \overline{deliver}\,[u, v] \mid \overline{record}\,[u, v]) \end{array}$$

We note that several clients may compete for the same item. In this case, delivery occurs only when the payment for the item is available. We also observe that the channel $voucher$ is co-located with $item$. Henceforth, the orchestrator $voucher(u) \mathbin{\&} item() \triangleright \overline{deliver}\,[u, v] \mid \overline{record}\,[u, v]$ will coordinate two channel managers at a same location.

Synchronizing merge. *Synchronizing merge* is one of the advanced synchronization patterns in [14]. According to this pattern, an activity A may trigger one or two concurrent activities B and C. These activities B and C signal their completion by sending messages to a fourth activity D. Synchronization occurs only if both B and C have been triggered. We assume that the choice of A of triggering one between B and C or both is manifested to D by emitting a signal over one or not, respectively. This signal is similar to the so-called "false tokens" in those workflow engines that support this synchronization pattern:

$$SynchMerge \stackrel{\text{def}}{=} b(v) \,\&\, one() \triangleright \overline{d}\,[v] + c(v) \,\&\, one() \triangleright \overline{d}\,[v] + b(v) \,\&\, c(v') \triangleright \overline{d}\,[v, v']$$

Dynamic load balancing. Our last example models a load balancing mechanism for Web services. We assume the existence of two message queues: job, where requests for services are posted, $ready$ where services make themselves available for processing one or more requests:

$$LoadBal \stackrel{\text{def}}{=} \ldots\, ready(w) \,\&\, job(u) \triangleright \overline{w}\,[u]$$

This is a typical load balancing mechanism that can handle multiple requests concurrently, or may distribute the computational load among different servers, possibly depending on request's priority. In addition, in our language it is possible to change the load dynamically. For instance, a supplier could run the code

$$job(u) \,\&\, job(v) \triangleright \overline{w}\,[u, u']$$

that changes the policy by processing two jobs at a same time. This small piece of code – a smooth orchestrator – may migrate to the location of the load balance process in order to update the current policy.

4 The Smoothness Restriction

A distributed prototype of the calculus in Section 2 may be designed with difficulties. Let us commit to a standard abstract machine of several distributed implementations of process calculi [15, 6, 4]. Such a machine consists of processors running at different locations with channels that are uniquely located to processors. Outputs are always delivered to the processor of the corresponding subject where they may be consumed. In this machine, the process

$$x(u \mathbin{@} u, v \mathbin{@} v) \;\triangleright\; \big(u(w) \,\&\, v(w') \triangleright P\big)$$

dynamically creates an orchestrator on the received channels u and v. There are at least two problematic issues as far as distribution is concerned. Let z and y be the names respectively replacing u and v at run-time:

1. the orchestrator $z(w) \,\&\, y(w') \triangleright P$ is consuming inputs on channels z and y that may be not co-located. This means that the processors running such channel(-manager)s must compete with the processor running $z(w) \,\&\, y(w') \triangleright P$ for consuming output. This is a classical global consensus problem.

2. if the channels z and y were co-located, the global consensus problem could be solved by migrating the orchestrator $z(w) \,\&\, y(w') \triangleright P$ to the right processor. However, this migration is expensive, because P may be large and could require a large closure.

Due to these problems, it is preferable to restrain our study to a sub-calculus of that in Section 2, which is more amenable to distributed implementations. The restrictions we consider are the following two:

1. every orchestrator is *smooth*, namely it has the shape $\sum_{i\in I} J_i \triangleright \overline{z_i}\,\widetilde{u}_i$ where $\widetilde{u}_i$ is the sequence of bound names in J_i in the same order as they appear in J_i and $\bigcup_{i\in I} \mathtt{sn}(J_i)$ are all co-located;
2. we admit processes $z(\widetilde{u} \mathbin{@} \widetilde{v}) \triangleright P$, namely generic continuations are restricted to simple inputs.

The formalization of the co-location restriction of joins in smooth orchestrators is defined by means of the type system in Table 1. Let ε denote the empty co-location sequence.

Table 1. Co-location checks for the full calculus

$$\text{(NIL)}\quad \Lambda \vdash 0 \qquad\qquad \text{(OUTPUT)}\quad \Lambda \vdash \overline{x}\,\widetilde{u}$$

$$\text{(ORCH)}\ \frac{(\vdash J_i :: \Lambda_i \quad \Lambda\Lambda_i \vdash P_i)^{i\in I} \quad (\Lambda \vdash x^\frown y)^{x\in \mathtt{sn}(J_i), y\in \mathtt{sn}(J_j)}}{\Lambda \vdash \sum_{i\in I} J_i \triangleright P_i} \qquad \text{(NEW)}\ \frac{\Lambda(x \mathbin{@} y) \vdash P}{\Lambda \vdash (x \mathbin{@} y)P}$$

$$\text{(PAR)}\ \frac{\Lambda \vdash P \quad \Lambda \vdash Q}{\Lambda \vdash P \mid Q} \qquad \text{(BANG)}\ \frac{\Lambda \vdash P}{\Lambda \vdash\, !P} \qquad \text{(INPUT)}\ \vdash x(\widetilde{u} \mathbin{@} \widetilde{v}) :: \widetilde{u} \mathbin{@} \widetilde{v} \qquad \text{(JOIN)}\ \frac{\vdash J :: \Lambda' \quad \vdash J' :: \Lambda''}{\vdash J \,\&\, J' :: \Lambda'\Lambda''}$$

Definition 5. *A process P is* distributable *if $\varepsilon \vdash P$.*

We defer the analysis of the distributed implementation of smooth orchestrators to Section 5. The rest of the section is devoted to the correctness of the co-location system and the encoding of the calculus in Section 2 into the sub-calculus with smooth orchestrators. We begin with a couple of technical statements.

Lemma 1. *1. Let Λ and Λ' be such that, for every $x, y \in \mathtt{fn}(P)$, if $\Lambda \vdash x^\frown y$ then $\Lambda' \vdash x^\frown y$. Then $\Lambda \vdash P$ implies $\Lambda' \vdash P$.*

2. Let x' be fresh with respect to names in Λ and in $\mathtt{fn}(P)$. Then $\Lambda(x \mathbin{@} y) \vdash P$ implies $\Lambda(x' \mathbin{@} y\{x'/x\}) \vdash P\{x'/x\}$.

Proof. We prove item 2; the first is simpler. The argument is by induction on the proof of $\Lambda(x \mathbin{@} y) \vdash P$ and we discuss the case when the last rule is an instance of (NEW); the others are similar or straightforward.

In this case $P = (u \mathbin{@} v)P'$ and $\Lambda(x \mathbin{@} y) \vdash (u \mathbin{@} v)P'$. By (NEW) one reduces to $\Lambda(x \mathbin{@} y)(u \mathbin{@} v) \vdash P'$. There are two sub-cases (a) $u \neq x, y$ and $v \neq x$ and (b) the others. In (a), by Proposition 1 and item 1, we also have $\Lambda(u \mathbin{@} v)(x \mathbin{@} y) \vdash P'$. Then, by inductive hypotheses, it is possible to obtain $\Lambda(u \mathbin{@} v)(x' \mathbin{@} y\{^{x'}/_{x}\}) \vdash P'\{^{x'}/_{x}\}$. Using again item 1 we derive $\Lambda(x' \mathbin{@} y\{^{x'}/_{x}\})(u \mathbin{@} v) \vdash P'\{^{x'}/_{x}\}$ and we conclude by (NEW). The sub-case (b) is proved as follows. Let $\Lambda(x \mathbin{@} y)(u \mathbin{@} v) \vdash P'$. The clashes of u with x or y may be removed by inductive hypothesis. Therefore, the problematic case is $\Lambda(x \mathbin{@} y)(u \mathbin{@} x) \vdash P'$.

If $x \neq y$ then consider the co-location sequence $\Lambda(u \mathbin{@} y)(x \mathbin{@} y)$. It is easy to prove that $\Lambda(x \mathbin{@} y)(u \mathbin{@} x) \vdash u'^{\frown}v'$ implies $\Lambda(u \mathbin{@} y)(x \mathbin{@} y) \vdash u'^{\frown}v'$. Therefore, by item 1 we may derive $\Lambda(u \mathbin{@} y)(x \mathbin{@} y) \vdash P$ and, by inductive hypothesis, we derive $\Lambda(u \mathbin{@} y)(x' \mathbin{@} y\{^{x'}/_{x}\}) \vdash P\{^{x'}/_{x}\}$. With a same argument it is possible to obtain $\Lambda(x' \mathbin{@} y\{^{x'}/_{x}\})(u \mathbin{@} x') \vdash P\{^{x'}/_{x}\}$ and by (NEW) we conclude $\Lambda(x' \mathbin{@} y\{^{x'}/_{x}\}) \vdash (u \mathbin{@} x')P\{^{x'}/_{x}\}$.

If $x = y$ then we consider the co-location sequence $\Lambda(u \mathbin{@} u)(x \mathbin{@} u)$. The proof may be completed as in the previous sub-case. □

It is worth to notice that Lemma 1 entails the weakening statement: if $\Lambda \vdash P$ and x is fresh then $\Lambda(x \mathbin{@} y) \vdash P$.

Lemma 2 (substitution). *Let ρ be a substitution such that $\mathtt{dom}(\rho) = \widetilde{x}$. If $\Lambda(\widetilde{x} \mathbin{@} \widetilde{y}) \vdash P$ and $\Lambda \vdash \widetilde{x}\rho^{\frown}\widetilde{y}\rho$ then $\Lambda \vdash P\rho$.*

Proof. The argument is by induction on the proof of $\Lambda(\widetilde{x} \mathbin{@} \widetilde{y}) \vdash P$. The interesting cases are when the last rule is an instance of (NEW) and of (ORCH).

(NEW). Let $P = (u \mathbin{@} v)P'$ and $u \neq v$ (the case $u = v$ is similar). By (NEW) we are reduced to

$$\Lambda(\widetilde{x} \mathbin{@} \widetilde{y})(u \mathbin{@} v) \vdash P' \tag{1}$$

There are a number of sub-cases:

- $u \notin \widetilde{x}\widetilde{y}$ and $v \notin \widetilde{x}$. Then by Proposition 1 the contexts $\Lambda(\widetilde{x} \mathbin{@} \widetilde{y})(u \mathbin{@} v)$ and $\Lambda(u \mathbin{@} v)(\widetilde{x} \mathbin{@} \widetilde{y})$ are equivalent and by item 1 we have $\Lambda(u \mathbin{@} v)(\widetilde{x} \mathbin{@} \widetilde{y}) \vdash P'$. By applying the inductive hypothesis and (NEW) we obtain $\Lambda \vdash (u \mathbin{@} v)(P\rho)$ that leads to $\Lambda \vdash ((u \mathbin{@} v)P)\rho$ since $u, v \notin \widetilde{x}$.
- $u \notin \widetilde{x}\widetilde{y}$ and $v \in \widetilde{x}$. We discuss two possibilities. If $(v \mathbin{@} w) \in (\widetilde{x} \mathbin{@} \widetilde{y})$ and $w \notin \widetilde{x}$ the contexts $\Lambda(\widetilde{x} \mathbin{@} \widetilde{y})(u \mathbin{@} v)$ and $\Lambda(\widetilde{x} \mathbin{@} \widetilde{y})(u \mathbin{@} w)$ are equivalent and we can apply the same arguments as for the previous case. If $(v \mathbin{@} v) \in (\widetilde{x} \mathbin{@} \widetilde{y})$ consider the context $\Lambda(u \mathbin{@} v\rho)(\widetilde{x}' \mathbin{@} \widetilde{y}')$ where $(\widetilde{x}' \mathbin{@} \widetilde{y}')$ is obtained by substituting $(v \mathbin{@} v)$ with $(v \mathbin{@} u)$ in $(\widetilde{x} \mathbin{@} \widetilde{y})$. Note that $\Lambda(\widetilde{x} \mathbin{@} \widetilde{y})(u \mathbin{@} v) \vdash u'^{\frown}v'$ implies $\Lambda(u \mathbin{@} v\rho)(\widetilde{x}' \mathbin{@} \widetilde{y}') \vdash u'^{\frown}v'$ so we can apply item 1 followed by the inductive hypothesis and obtain $\Lambda(u \mathbin{@} v\rho) \vdash P'\rho$. From this we derive $\Lambda \vdash (u \mathbin{@} v\rho)P'\rho$, which is equivalent to $\Lambda \vdash ((u \mathbin{@} v)P')\rho$.
- $u \in \widetilde{x}\widetilde{y}$. By Lemma 1(2) we reduce to $\Lambda(\widetilde{x} \mathbin{@} \widetilde{y})(u' \mathbin{@} v) \vdash P'\{^{u'}/_{u}\}$, where u' is fresh, therefore $u' \notin \widetilde{x}\widetilde{y}$. In the same way as in the first sub-case, we obtain $\Lambda \vdash ((u' \mathbin{@} v)P'\{^{u'}/_{u}\})\rho$ and we conclude because, by definition of substitution, $((u' \mathbin{@} v)P'\{^{u'}/_{u}\})\rho = ((u \mathbin{@} v)P')\rho$ when $u \in \widetilde{x}\widetilde{y}$.

(ORCH). We discuss the case when $P = J \triangleright P'$. The general case is similar. By (ORCH), and letting $\vdash J :: (\widetilde{z} \mathbin{@} \widetilde{w})$ we are reduced to

$$\Lambda(\widetilde{x} \mathbin{@} \widetilde{y})(\widetilde{z} \mathbin{@} \widetilde{w}) \vdash P' \tag{2}$$

and

$$(\Lambda(\widetilde{x} \mathbin{@} \widetilde{y}) \vdash x' \frown y')^{x', y' \in \mathtt{sn}(J)} \tag{3}$$

We may apply Proposition 2 to (3) and obtain

$$(\Lambda \vdash x'\rho \frown y'\rho)^{x', y' \in \mathtt{sn}(J)} \tag{4}$$

From (2), with a similar argument as in (NEW), we may derive $\Lambda(\widetilde{z} \mathbin{@} \widetilde{w}\rho) \vdash P'\rho$ or similar judgments renaming names in $\widetilde{z}$ when they clash with $\widetilde{x}\widetilde{y}$ (we omit these last cases). By applying (ORCH) to this judgment, to (4) and to $\vdash J\rho :: (\widetilde{z} \mathbin{@} \widetilde{w}\rho)$ we therefore derive $\Lambda \vdash J\rho \triangleright P'\rho$. We conclude by observing that $J\rho \triangleright P'\rho = (J \triangleright P')\rho$. □

A brief discussion about the substitution lemma follows. Consider

$$(a \mathbin{@} a)(u \mathbin{@} u)(v \mathbin{@} u) \vdash u \,\&\, v \triangleright \mathsf{0}$$

and the substitution $\{^{a}/_{v}\}$. Note that $(a \mathbin{@} a)(u \mathbin{@} u) \not\vdash v\{^{a}/_{v}\} \frown u\{^{a}/_{v}\}$. Indeed, if we were allowed to apply $\{^{a}/_{v}\}$ to the above judgment, we would obtain

$$(a \mathbin{@} a)(u \mathbin{@} u) \not\vdash u \,\&\, a \triangleright \mathsf{0}$$

Actually, the process $u \,\&\, v \triangleright \mathsf{0}$ is well-typed in a context that co-locates u and v. While this is the case for $(a \mathbin{@} a)(u \mathbin{@} u)(v \mathbin{@} u)$, it is not the case for $(a \mathbin{@} a)(u \mathbin{@} u)$. The condition $\Lambda \vdash \widetilde{x}\rho \frown \widetilde{y}\rho$ in the substitution establishes that co-located names remain co-located after having been substituted. Therefore, if we insist in replacing v with a, we must also map u to a. In this case the substitution lemma may be applied and we obtain:

$$(a \mathbin{@} a) \vdash a \,\&\, a \triangleright \mathsf{0}$$

Theorem 1 (subject reduction). *If $(\widetilde{x} \mathbin{@} \widetilde{y}) \vdash P$ and $(\widetilde{x} \mathbin{@} \widetilde{y})P \to (\widetilde{x} \mathbin{@} \widetilde{y})Q$ then $(\widetilde{x} \mathbin{@} \widetilde{y}) \vdash Q$. In particular, if P is distributable and $P \to Q$ then Q is distributable as well.*

Proof. It is sufficient to show that well-typedness is preserved by any structural congruence rule (in both directions) and by the reduction rule. We omit the easy cases.

- Let $\Lambda \vdash (x \mathbin{@} y)(x' \mathbin{@} y')P$ with $x \neq y'$, $x \neq x'$, and $y \neq x'$. By (NEW): $\Lambda(x \mathbin{@} y)(x' \mathbin{@} y') \vdash P$. By Lemma 1(1): $\Lambda(x' \mathbin{@} y')(x \mathbin{@} y) \vdash P$. We conclude by (NEW).
- Let $\Lambda \vdash (x \mathbin{@} y)(P \mid Q)$ and $x \notin \mathtt{fn}(P)$. It is sufficient to show that if $u, v \neq x$ then $\Lambda \vdash u \frown v$ iff $\Lambda(x \mathbin{@} y) \vdash u \frown v$. This follows by the rule (LIFT) of the co-location relation.

– Let $\Lambda \vdash (M\rho \mid \sum_{i\in I} J_i \triangleright P_i \mid R)$ and let $(\Lambda)(M\rho \mid \sum_{i\in I} J_i \triangleright P_i \mid R) \to (\Lambda)(P_k\rho \mid R)$. By the hypotheses of the reduction rule: $M = \prod_{j=1..n} \overline{x_j}\, u_j$, $J_k = \&_{j\in 1..n} x_j(\widetilde{u}_j \mathbin{@} \widetilde{v}_j)$, $\mathtt{dom}(\rho) = \bigcup_{j=1..n} \widetilde{u}_j$ and $\Lambda \vdash \widetilde{u}_j\rho \frown \widetilde{v}_j\rho$ for all $j \in 1..n$. The type system yields $\vdash J_k :: (\tilde{u}_j \mathbin{@} \tilde{v}_j)^{j\in 1..n}$. Therefore, by the Substitution Lemma applied to $\Lambda(\tilde{u}_j \mathbin{@} \tilde{v}_j)^{j\in 1..n} \vdash P_k$, we obtain $\Lambda \vdash P_k\rho$. From this we conclude $\Lambda \vdash P_k\rho \mid R$. □

The calculus with distributable orchestrators may be encoded into the calculus with smooth ones. We first define an encoding that decouples complex continuations from join patterns.

Definition 6. *The encoding $[\![\cdot]\!]$ is defined on processes in Section 2. The function $[\![\cdot]\!]$ is an homomorphism except for orchestrators. In the definition below we assume that, for every j, $z_j \notin \bigcup_{i\in I}(\mathtt{fn}(J_i) \cup \mathtt{bn}(J_i))$ and the tuple $\widetilde{u}_j$ is exactly the sequence of defined names in J_j:*

$$[\![\sum_{i\in I} J_i \triangleright P_i]\!] \;=\; (z_i{}^{i\in I})\Big(\sum_{i\in I} J_i \triangleright \overline{z_i}\,\widetilde{u}_i \;\mid\; z_i(\widetilde{u}_i) \triangleright [\![P_i]\!]\Big)$$

It is folklore to demonstrate the correctness of the encoding $[\![\cdot]\!]$, namely $P \approx Q$ if and only if $[\![P]\!] \approx [\![Q]\!]$. This is an immediate consequence of the following statement, that in turn uses a generalization of the pi calculus law $x(\widetilde{u}).P \approx (z)(x(\widetilde{u}).\overline{z}\,\widetilde{u} \mid z(\widetilde{u}).P)$.

Proposition 3. *For every P, $P \approx [\![P]\!]$.*

Of course, if P is a generic process with orchestrators then join patterns in $[\![P]\!]$ may have subjects that are not co-located. It is possible to avoid such problematic cases by restricting the domain of $[\![\cdot]\!]$ to distributable processes.

Proposition 4. *If P is distributable then $[\![P]\!]$ is a process with smooth orchestrators.*

5 The Implementation of Smooth Orchestrators

Smooth orchestrators are small pieces of code that may migrate over the network for reaching the location where they execute. Unlike mobile agents, they exhibit a simple, finite behavior and they require a limited-size message to migrate. Consider a single branch smooth orchestrator:

$$x_1(\widetilde{u}_1 \mathbin{@} \widetilde{v}_1) \,\&\, \cdots \,\&\, x_n(\widetilde{u}_n \mathbin{@} \widetilde{v}_n) \triangleright \overline{z}\,\widetilde{u}_1 \cdots \widetilde{u}_n$$

It may be encoded as a vector of $n+1$ names – the subjects $x_1, \cdots, x_n$ plus the destination channel z – and a vector of $k_1 + \cdots + k_n$ values, where k_i is the length of the tuple $\widetilde{u}_i$. Each value can be either an integer or a (free) name and it encodes a co-location constraint: (1) the integer value j at position h indicates that the j-th and h-th bound names must be co-located; (2) the constant c at position h indicates that the h-th bound name must be co-located with c. An orchestrator of m branches is encoded by a vector of length m whose elements are pairs of vectors of the above shape.

The destination of this vector is driven by the location of the subjects (remember that the subjects are co-located). When this vector arrives at destination, it triggers an appropriate process that monitors the states of the message queues of the subjects. We discuss the implementation of smooth orchestrators of increasing complexity, starting from the automata-based technique for implementing join patterns in the join calculus, and gradually extending the technique to include the new features. Initially we omit the discussion of nonlinear patterns and orchestrators with multiple branches.

In the join calculus a join definition is compiled as a finite state automaton that keeps track of the status of the message queues associated with the corresponding channels [9]. Formally, let $x_1(\widetilde{u}_1) \,\&\, \cdots \,\&\, x_n(\widetilde{u}_n) \triangleright \overline{z}\,\widetilde{u}_i$ be the definition, which is also a smooth orchestrator. The associated automaton is

$$M = (\wp(\{x_1, \ldots, x_n\}), \{+x_1, -x_1, \ldots, +x_n, -x_n\}, \delta, \emptyset, \{x_1, \ldots, x_n\}) \qquad (5)$$

where

$$\delta(q, +x_i) = q \cup \{x_i\} \qquad \delta(q, -x_i) = q \setminus \{x_i\}$$

The automaton reacts to symbols of the form $+x$, meaning "the message queue of the channel x is not empty" and $-x$, meaning "the message queue of the channel x is empty". Every time a message queue changes (either because a new message arrives, or because a message is removed) it notifies all the automata associated with it. When the joined channels are all fresh (and join definitions cannot be extended at runtime, like in the join calculus) there will be a unique automaton for handling the whole definition. In our case channel orchestrations may be added and/or removed at runtime, thus making the set of automata associated with them change over time. The consequent competitions for messages in shared channel queues are solved without difficulties because the automata are all co-located.

This mechanism may be easily extended with co-location constraints when the scope of such constraints is limited to the message itself. This is the case in $x(u \mathbin{@} u, v \mathbin{@} u) \,\&\, y(w \mathbin{@} w) \triangleright P$. To model this extension the alphabet of the automata is patched by admitting symbols of the form $+x(\widetilde{u} \mathbin{@} \widetilde{v})$ and $-x(\widetilde{u} \mathbin{@} \widetilde{v})$ instead of $+x$ and $-x$. When a new message $\overline{x}\,\widetilde{a}$ is available, each automaton associated with x makes a $+x(\widetilde{u} \mathbin{@} \widetilde{v})$ transition only if the co-location constraints are satisfied. When a message $\overline{x}\,\widetilde{a}$ is removed from the x-queue, every automaton that has been affected by the message checks whether the queue contains another message satisfying its co-location constraints or not. In case there is no such message, the state of the automaton is adequately reset in accordance with the new state of the queue.

When different joined channels have co-location dependencies, the constraints to be verified may involve names that have been bound during previous transitions. For example, take $x(u \mathbin{@} u) \,\&\, y(v \mathbin{@} u)$ and assume that the corresponding automaton has made a transition on a message $\overline{x}\,a$. The subsequent transition on y depends on a's location: only a message $\overline{y}\,b$ such that a and b are co-located will make the automaton move into the accepting state. Symmetrically, the automaton may start with a message $\overline{y}\,b$. In this case the automaton may progress provided a message $\overline{x}\,a$ has been enqueued with a co-located with b. In facts, we are rewriting the above pattern into $y(v \mathbin{@} v) \,\&\, x(u \mathbin{@} v)$, which preserves the co-location constraints (this operation is sustained by Lemma 1.1)

and is left-constraining. More precisely, in the case of $J = \&_{i\in 1..n} x_i(\widetilde{u}_i \mathbin{@} \widetilde{v}_i)$ the automaton is defined as follows. Let $W = \bigcup_{j\in 1..n} \widetilde{u}_j\widetilde{v}_j$ and $\widetilde{w}_j$ be tuples in W. Then

$$M = (\{\, \&_{i\in I} x_i(\widetilde{u}_i \mathbin{@} \widetilde{w}_i) \mid I \subseteq 1..n\}, \{x_i(\widetilde{u}_i \mathbin{@} \widetilde{w}_i) \mid i \in 1..n\}, \delta, J, \emptyset)$$

where the transition relation δ is defined by

$$\Big(\&_{i\in I} x_i(\widetilde{u}_i \mathbin{@} \widetilde{w}_i)\Big) \,\&\, x(\widetilde{u} \mathbin{@} \widetilde{w}) \,\&\, \Big(\&_{j\in I'} x_j(\widetilde{u}_j \mathbin{@} \widetilde{w}_j)\Big) \xrightarrow{x(\widetilde{u} \mathbin{@} \widetilde{w}')} \Big(\&_{i\in I} x_i(\widetilde{u}_i \mathbin{@} \widetilde{w}'_i)\Big) \,\&\, \Big(\&_{j\in I'} x_j(\widetilde{u}_j \mathbin{@} \widetilde{w}_j)\Big)$$

where $(\widetilde{u}_i \mathbin{@} \widetilde{w}_i)^{i\in I}(\widetilde{u} \mathbin{@} \widetilde{w})$ is equivalent (in the sense of Lemma 1.1) to the sequence $(\widetilde{u} \mathbin{@} \widetilde{w}')(\widetilde{u}_i \mathbin{@} \widetilde{w}'_i)^{i\in I}$. (This rewriting can always be accomplished with simple syntactic transformations because the join pattern is left constrained.)

The instantaneous description of an automaton is a pair (q, ρ) where q is the current state and ρ is the substitution over names that have been bound while the automaton moved from the initial state to q. The behavior of the automaton can be defined by the following transition relation between instantaneous descriptions:

$$(q, \rho) \xrightarrow{\overline{x}\, a_1,\ldots,a_n} (q', \rho[u_1 \mapsto a_1] \cdots [u_n \mapsto a_n])$$

if $q' = \delta(q, x(u_1 \mathbin{@} v_1, \ldots, u_n \mathbin{@} v_n))$ and $a_i \frown v_i\rho$. Note that the behavior is not deterministic: an incoming message may spawn a new automaton at any time.

As usual this nondeterminism may be described in terms of multiple automata running simultaneously, or by means of backtracking when there is a choice. It is well known that nondeterministic automata are considerably more expensive than the deterministic ones in terms of space occupation or computational complexity. Since this complexity is unavoidable if constraints make two or more input channels depend on each other, it makes sense to look for solutions that limits the use of nondeterministic automata as much as possible. One of such solutions that we consider is the following. Given a pattern $J = \&_{i\in I} x_i(\widetilde{u}_i \mathbin{@} \widetilde{v}_i)$ we can partition the set of x_i's so that two channels stay in the same partition only if they have co-location dependencies. Inputs that only have local co-location constraints, like $x(u \mathbin{@} u)$ or $y(u \mathbin{@} u, v \mathbin{@} u)$, are placed in singleton partitions. Then, a deterministic automaton can be created for handling the pattern J partition-wise. On the contrary, every partition that contains inputs with co-location dependencies will be implemented by means of a nondeterministic automaton. It turns out that this simple optimization is effective since most of the orchestrators with complex join patterns that are used in practice have very few co-location dependencies.

So far the implementation of smooth orchestrators that are not linear with respect to subjects have been purposely overlooked. The deterministic automaton 5 described above can handle nonlinear patterns following the suggestion of Maranget and Le Fessant in [9]. The basic observation is that the number of channels involved in a pattern is finite and the automata can query the associated message queues for the number of the needed messages. Because of their nature, nondeterministic automata can also handle nonlinear patterns. On the contrary, deterministic automata with co-location constraints cannot be extended in a straightforward way. Consider the pattern $x(u \mathbin{@} u, v \mathbin{@} v) \,\&\, x(w \mathbin{@} w, z \mathbin{@} w)$. If a message satisfies $x(w \mathbin{@} w, z \mathbin{@} w)$, but the automaton uses it for

making a transition on $x(u \mathbin{@} u, v \mathbin{@} v)$, then it might be not possible to reach the accepting state. What is needed in this case is again a form of nondeterminism.

The implementation of orchestrators that consist of several branches makes use of the solution adopted in join calculus that mostly *merges* the automata for different branches into a single automaton. The idea being that if the branches involve shared inputs, the automata usually share some common structure and the resulting automaton is smaller than the sum of the automata for the branches taken separately.

References

1. Aceto, L., Bloom, B., Vaandrager, F.W.: Turning SOS rules into equations. Information and Computation **111**(1) (1994) 1–52
2. Andrews, T., et.al.: Business Process Execution Language for Web Services. Version 1.1. Specification, BEA Systems, IBM Corp., Microsoft Corp., SAP AG, Siebel Systems (2003)
3. Busi, N., Padovani, L.: A distributed implementation of mobile nets as mobile agents. In: Proceedings of the 7th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2005). Volume 3535 of LNCS., Springer Verlag (2005) 259–274
4. Brown, A., Laneve, C., Meredith, G.L.: PiDuce: a process calculus with native XML datatypes. In: 2nd International Workshop on Web Services and Formal Methods (WS-FM 2005). Volume 3670 of LNCS., Springer Verlag (2005) 18–34
5. Carpineti, S., Laneve, C., Milazzo, P.: The BoPi machine: a distributed machine for experimenting web services technologies. In: Fifth International Conference on Application of Concurrency to System Design (ACSD 2005), IEEE Computer Society Press (2005) 202–211
6. Fournet, C.: The Join-Calculus: A Calculus for Distributed Mobile Programming. PhD thesis, École Polytechnique, Paris, France (1998)
7. Gardner, P., Laneve, C., Wischik, L.: Linear forwarders. In R. Amadio, D.L., ed.: CONCUR 2002. Volume 2761 of Lecture Notes in Computer Science., Springer-Verlag (2003) 415–430
8. Kavantzas, N., Olsson, G., Mischkinsky, J., Chapman, M.: Web Services Choreography Description Languages. Oracle Corporation (2003)
9. Le Fessant, F., Maranget, L.: Compiling join-patterns. In Nestmann, U., Pierce, B.C., eds.: Proceedings of High-Level Concurrent Languages '98. Volume 16.3 of Electronic Notes in Computer Science. (1998)
10. Leymann, F.: Web Services Flow Language (wsfl 1.0). Technical report, IBM Software Group (2001)
11. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, part I/II. Information and Computation **100** (1992) 1–77
12. Milner, R., Sangiorgi, D.: Barbed bisimulation. In: Proceedings of ICALP '92. Volume 623 of Lecture Notes in Computer Science., Springer-Verlag (1992) 685–695
13. Thatte, S.: XLANG: Web services for business process design. Microsoft Corporation (2001)
14. van der Aalst, W.: Workflow patterns. At `www.workflowpatterns.com` (2001)
15. Wojciechowski, P., Sewell, P.: Nomadic pict: Language and infrastructure design for mobile agents. In: Proceedings of ASA/MA '99 (First International Symposium on Agent Systems and Applications/Third International Symposium on Mobile Agents). (1999)