

Orchestrating Tuple-based Languages[★]

Rocco De Nicola^{1,2}, Andrea Margheri¹, and Francesco Tiezzi²

¹ Univeristà degli Studi di Firenze, Dipartimento di Sistemi e Informatica

² IMT - Institute for Advanced Studies Lucca

`{rocco.denicola,francesco.tiezzi}@imtlucca.it, margheri.a@alice.it`

Abstract. The World Wide Web can be thought of as a global computing architecture supporting the deployment of distributed networked applications. Currently, such applications can be programmed by resorting mainly to two distinct paradigms: one devised for orchestrating distributed services, and the other designed for coordinating distributed (possibly mobile) agents. In this paper, the issue of designing a programming language aiming at reconciling orchestration and coordination is investigated. Taking as starting point the orchestration calculus ORC and the tuple-based coordination language KLAIM, a new formalism is introduced combining concepts and primitives of the original calculi. To demonstrate feasibility and effectiveness of the proposed approach, a prototype implementation of the new formalism is described and it is then used to tackle a case study dealing with a simplified but realistic electronic marketplace, where a number of on-line stores allow client applications to access information about their goods and to place orders.

Keywords: Global computing, Orchestration, Coordination, Tuple-based languages, Formal methods, Software tools

1 Introduction

In recent years, the growing success of e-business, e-learning, e-government, and similar emerging models, has led the World Wide Web, initially thought of as a tool supporting humans in looking for information, to evolve towards a service-oriented architecture, where more and more networked applications, the so-called *services*, are deployed. This has promoted the rising of a novel programming paradigm for the *orchestration* of concurrent and distributed services. There are by now some successful and well-developed technologies supporting this paradigm, like e.g. WS-BPEL [33], the standard language for orchestration of web services. However, current software engineering technologies remain at the descriptive level and lack rigorous formal foundations. Hence, many researchers have tackled the problem at a more foundational level, by developing formal languages for designing and programming service orchestrations.

Among the many proposed formalisms (see, e.g., [28, 8, 10, 23, 11, 7]), we will focus on ORC [31, 40], a task orchestration language with applications in workflow, business process management, and web service orchestration. ORC is the

[★] This work has been partially sponsored by the EU project ASCENS, 257414.

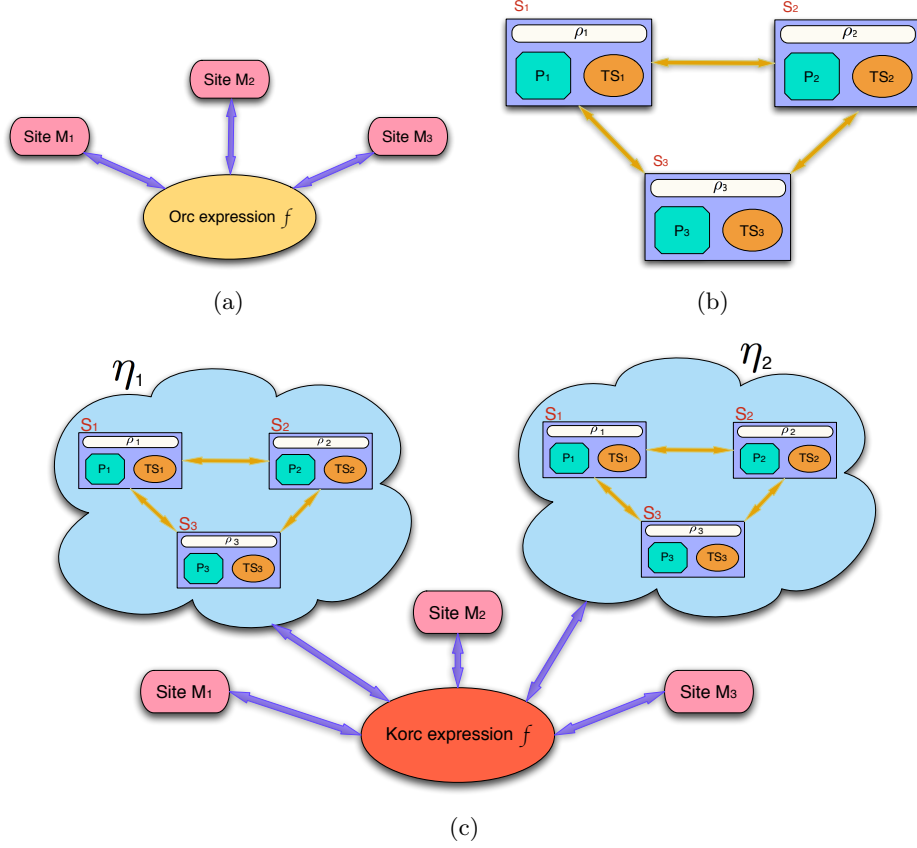


Fig. 1. The ORC (a), KLAIM (b) and KORC (c) approaches

result of a tension between simplicity and expressiveness, and its primitives, differently from most the formalisms mentioned above, focus on orchestration rather than on communication. An ORC program, graphically depicted in Figure 1(a), is an *expression* that orchestrates concurrent invocations of a number of services, called *sites* in the ORC's jargon, by means of three operators modelling sequential and parallel composition.

The small numbers of ORC's operators have been proved to be sufficiently expressive to model the most common orchestration patterns (e.g. those identified in [38]). However, they do not provide adequate and flexible mechanisms for distributed *coordination*, which may possibly refer and exploit the structures of the network. *Tuple-based* languages have, instead, been effectively used to implement coordination mechanism in a distributed setting. Among the many proposals (see, e.g., [22, 37, 35, 17, 14, 36]), here, we would like to focus on KLAIM [15, 6, 16], a coordination language specifically designed to program distributed systems consisting of mobile components interacting through multiple distributed tuple spaces. The KLAIM's communication model builds over, and extends, Linda's notion of generative communication through a single shared tuple space [22] and

(Expressions)	$f, g ::= M(\bar{p}) \mid E(\bar{p}) \mid f > \bar{p} > g \mid f \mid g \mid f < \bar{p} < g$
(Parameters)	$p ::= x \mid m$

Table 1. ORC syntax

its primitives allow programmers to distribute and retrieve data and processes to and from the nodes of a net. Localities are first-class citizens that can be dynamically created and communicated over the network and can be handled via sophisticated scoping rules.

A KLAIM specification, graphically depicted in Figure 1(b), can be thought of as a net of interconnected nodes, each of which hosts data tuples and (possibly mobile) processes, and is identified by a unique name.

In this paper, we investigate the issue of designing a programming language aiming at reconciling the orchestration paradigm with the tuple-based coordination one and define a new formalism, called KORC, that combines composition patterns and primitives of ORC and KLAIM. Intuitively, a KORC program, graphically depicted in Figure 1(c), consists of an ORC expression and a collection of KLAIM nets. Expressions are enriched with primitives for acting on the tuple spaces within the KLAIM nets, the latter are named and can be referred within the expressions.

The choice of using ORC and KLAIM as theoretical basis for KORC has been mainly motivated by the fact that they are compact formalisms and are already supported by software tools for programming networked applications. Such tools are Java-based and, hence, easily integrable. Indeed, to demonstrate effectiveness of the programming paradigm fostered by KORC and to experiment with it, we have developed a prototype implementation of the new language that builds upon the implementations of ORC and KLAIM.

The rest of the paper is structured as follows. Section 2 presents the design and the formal definition of KORC, by introducing concepts and definitions of ORC and KLAIM. Section 3 introduces an e-commerce scenario that illustrates the relevant and specific aspects of KORC. Section 4 provides an overview of the prototype implementation of KORC and describes an excerpt of the e-commerce scenario written in the syntax accepted by the tool. Finally, Section 5 draws a few conclusions and reviews some strictly related work.

2 From Orc and Klaim to Korc

In this section, we first recap the basic notions of ORC and KLAIM, borrowed from [40] and [16], then we use them to define KORC.

Orc: an orchestration language. An ORC program consists of a goal *expression* and a set of *definitions*; the goal expression is evaluated in order to run the program. The definitions can be used in the expression and in other definitions. Formally, the ORC syntax is defined in Table 1, where M ranges over *site* names, E over *expression* names, x over variables, and m over values. It is assumed that

the sets of site names, expression names, variables and values are countable and pairwise disjoint.

Since we aim at merging ORC with a tuple-based coordination language, we consider the polyadic variant of ORC informally described in [26] that permits using tuples as parameters rather than single values. The overbar $\bar{\cdot}$ over a name denotes tuples of parameters, thus \bar{m} is the compact notation for the tuple of values $\langle m_1, \dots, m_n \rangle$ (with $n \geq 0$). Variables in the same tuple are pairwise distinct. The empty tuple, written $\langle \rangle$, corresponds to a *signal*, i.e. the ORC unit value that has no additional information. When convenient, we shall regard tuples simply as sets.

Expressions can be composed by means of sequential composition $\cdot > \bar{p} > \cdot$, symmetric parallel composition $\cdot \mid \cdot$, and asymmetric parallel composition $\cdot < \bar{p} < \cdot$, starting from site calls $M(\bar{p})$ and expression calls $E(\bar{p})$. The variables within \bar{p} are *bound* in g for the expressions $f > \bar{p} > g$ and $g < \bar{p} < f$. We use $\text{fv}(f)$ to denote the set of variables that are not bound (i.e. which occur *free*) in f . Each expression name E has a unique declaration of the form $E(\bar{x}) \triangleq f$, where $\bar{x} = \text{fv}(f)$, i.e. only the variables \bar{x} are free in f . The evaluation of an expressions may call a number of sites and returns, i.e. *publishes* in ORC's jargon, a (possibly empty) stream of (tuple of) values.

Informally, the semantics of ORC expressions is as follows:

- Site call:** a site call can have the form $M(\bar{p})$, where the site name M is known statically, and \bar{p} are the parameters of the call. A site call returns at most one response and, hence, a site might also not respond. If \bar{p} contains variables, then they must be instantiated before the call.
- Expression call:** an expression call has the form $E(\bar{p})$ and executes the expression defined by $E(\bar{x}) \triangleq f$ after replacing \bar{x} by \bar{p} in f (of course, the tuples \bar{x} and \bar{p} must have the same length). Here \bar{p} is passed by reference. Expression definitions can be recursive.
- Symmetric parallel composition:** the composition $f \mid g$ executes both f and g concurrently, assuming that there is no interaction between them. It publishes the interleaving of the two streams of tuples published by f and g .
- Sequential composition:** the composition $f > \bar{p} > g$ executes f and, for each tuple of values \bar{m} returned by f , it checks if \bar{p} and \bar{m} match. If this is the case, an instance of g is executed with variables in \bar{p} replaced by the corresponding values in \bar{m} . Otherwise the publication is ignored and no new instance of g is executed. The composition publishes the interleaving of the streams of tuples published by the different instances of g .
- Asymmetric parallel composition:** the composition $g < \bar{p} < f$ starts in parallel both f and the parts of g that do not need the variables in \bar{p} . When f publishes a tuple, let say \bar{m} , if \bar{p} and \bar{m} do match the evaluation of f terminates and the variables within \bar{p} are replaced by the corresponding values in \bar{m} (in this way, the suspended parts of g can proceed). The composition publishes the stream obtained from g (instantiated with values in \bar{m}).

More formally, the operational semantics of ORC is given in terms of a labelled transition relation and an auxiliary function for pattern-matching on semi-

(Nets)	$N ::= \mathbf{0}$		$s ::_{\rho} C$		$N_1 \parallel N_2$		$(\nu s)N$
(Components)	$C ::= P$		$\langle t \rangle$		$C_1 \mid C_2$		
(Processes)	$P ::= \mathbf{nil}$		$\alpha.P$		$P_1 \mid P_2$		$A(\bar{p})$
(Actions)	$\alpha ::= \mathbf{out}(t)@l$		$\mathbf{in}(T)@l$		$\mathbf{read}(T)@l$		
			$\mathbf{eval}(P)@l$		$\mathbf{newloc}(s)$		
(Tuples)	$t ::= e$		ℓ		P		t_1, t_2
(Templates)	$T ::= e$		ℓ		$!x$		$!l \mid !X \mid T_1, T_2$

Table 2. KLAIM syntax

structured data. Due to space limitations, we refer the interested reader to [29] for a full account of the ORC's operational semantics considered in this paper.

Klaim: a language for agents interaction and mobility. KLAIM is a formal language equipped with primitives for network-aware programming that combines a process algebraic approach with a coordination-oriented one. The syntax of KLAIM is reported in Table 2, where s, s', \dots range over *locality names* (i.e. network addresses); **self**, l, l', \dots range over *locality variables* (i.e. aliases for addresses); ℓ, ℓ', \dots range over locality names and variables; x, y, \dots range over *value variables*; X, Y, \dots range over *process variables*; e, e', \dots range over *expressions*³; A, B, \dots range over *process identifiers*⁴. We assume that the set of variables (i.e. locality, value and process variables), the set of values (locality names and basic values) and the set of process identifiers are countable and pairwise disjoint.

Nets are finite plain collections of nodes where *components*, i.e. processes and evaluated tuples, can be allocated. In the net $(\nu s)N$, the scope of the name s is restricted to N ; the intended effect is that if one considers the net $N_1 \parallel (\nu s)N_2$ then locality s of N_2 cannot be referred to from within N_1 .

A *node* is a triple $s ::_{\rho} C$, where the locality s is the address of the node, ρ is the allocation environment and C are the hosted components. An *allocation environment* binds the locality variables occurring free in the processes allocated in the corresponding node. Basically, allocation environments provide a name resolution mechanism by mapping locality variables l into localities s . The distinguished locality variable **self** is used by processes to refer to the address of their current hosting node.

Processes are the KLAIM active computational units. They are built up from the special process **nil**, which does not perform any action, and from the basic actions by means of action prefixing $\alpha.P$, parallel composition $P_1 \mid P_2$ and

³ The precise syntax of expressions is deliberately not specified, but we assume that they contain basic values (ranged over by v, v', \dots) and variables.

⁴ We assume that each process identifier A has a unique definition, visible from any locality of a net, of the form $A(\bar{f}) \triangleq P$, where the formal parameters in \bar{f} are pairwise distinct. Like for ORC, \bar{p} in the call $A(\bar{p})$ is the tuple of actual parameters.

(Expressions)	$f, g ::= M(\bar{p}) \mid E(\bar{p}) \mid f > \bar{p} > g \mid f \mid g \mid f < \bar{p} < g$
	$\mid \mathbf{out}(t)@ \eta : \ell \mid \mathbf{eval}(P)@ \eta : \ell$
	$\mid \mathbf{in}(T)@ \eta : \ell \mid \mathbf{read}(T)@ \eta : \ell$
(Named nets)	$\mathcal{K} ::= \{\eta_i ::_{\rho_i} N_i\}_{i \in I}$

Table 3. KORC syntax

process definition. Process may be executed concurrently either at the same locality or at different localities and can perform five different basic actions.

Actions **out**, **in** and **read** manage data repositories by adding/withdrawing/accessing data to/from node repositories. Action **eval** activates a new thread of execution, i.e. a process, in a (possibly remote) node. Action **newloc** permits creating new network nodes. All actions, apart for **newloc**, are indexed by the (possibly remote) locality where they will take place. Actions **in** and **read** are blocking actions and exploit templates as patterns to select data in shared repositories. *Templates* are sequences of actual and formal fields, where the latter are written $!x, !l$ or $!X$ and are used to bind variables to basic values, locality names or processes, respectively. Actions **out** and **eval** are non-blocking and implement *static* and *dynamic scoping* disciplines, respectively (see [16, 29]).

Names and variables occurring in KLAIM processes and nets can be *bound*. More precisely, prefix **newloc**(s). P binds name s in P , and, similarly, net restriction $(\nu s)N$ binds s in N . The sets $\text{fn}(\cdot)$ and $\text{bn}(\cdot)$ of, respectively, free and bound locality names of a term are defined accordingly. Prefixes **in**($\dots, !_, \dots$)@ $\ell.P$ and **read**($\dots, !_, \dots$)@ $\ell.P$ binds variable $_$ in P . A name/variable that is not bound is called *free*.

The operational semantics of KLAIM is given in terms of a structural congruence relation and a reduction relation expressing the evolution of nets. Due to space limitations, we refer the interested reader to [29] for a complete account of the KLAIM's semantics considered in this paper.

Korc: a language for orchestrating Klaim agents. We now show how the orchestration approach of ORC and the network-aware one of KLAIM can be combined in order to define a new formalism for orchestrating concurrent processes coordinated via distributed tuple spaces. More specifically, in this section we present the syntax and the operational semantics of the new calculus, that we call KORC.

A KORC program consists of a configuration (f, \mathcal{K}) , where f is an extended ORC expression (possibly equipped with a set of expression definitions) and \mathcal{K} is a set of *named KLAIM nets*. To execute a program, f is evaluated while the nets are concurrently running. The KORC syntax is defined in Table 3, where f is an ORC expression (like in Table 1) extended with KLAIM actions; η ranges over *net names*. Parameters p are defined in Table 1, and N , P , t and T are defined in Table 2. We assume that the KORC set of values, ranged over by m , includes the KLAIM set of values. Symbol \uplus is used to denote disjoint union of sets.

A KORC expression can interact with different KLAIM nets that can be referred (and distinguished) by means of net names. A *named net* is a triple $\eta ::_{\rho} N$, where η is the name of the net, ρ is the allocation environment used to bind location variables within KORC expressions, and N is a KLAIM net. Besides site and expression calls, a KORC expression can perform **out**, **eval**, **in** and **read** actions over named nets within the associated set \mathcal{K} . The actions have an additional argument η that explicitly indicates the target net. Action **newloc** cannot be directly executed by a KORC expression, because it only acts locally to a KLAIM node. However, it can be indirectly performed via **eval** actions.

A KORC program $(f, \{\eta_i ::_{\rho_i} N_i\}_{i \in I})$ is *well-formed* if names η_i are pairwise distinct and for each $i \in I$ we have that **self** is not in the domain of ρ_i and N_i is a well-formed net (see [16] and [29, Section 2.2]). Hereafter, we will only consider well-formed programs. Notably, we consider named nets, rather than unnamed ones, to avoid requiring locality names of all nets to be pairwise distinct. In fact, while this is reasonable when considering a single net, it becomes a too strong requirement in a distributed, loosely coupled, environment where different and independent subnets co-exist. The requirement on **self** is due to the fact that ρ_i are used to evaluate actions executed by a KORC expression and that, hence, are not hosted by any KLAIM node.

The operational semantics of KORC is given in terms of a labelled transition relation \xrightarrow{a} over configurations, which relies on the standard reduction relation \mapsto over KLAIM nets (see [29, Table 7]). As in the semantics of ORC, label a is generated by the following grammar:

$$a ::= \tau \quad | \quad !\bar{m}$$

Label τ indicates an *internal event*, while label $!\bar{m}$ indicates a *publication event* corresponding to the communication of the tuple of values \bar{m} after the evaluation of an expression. The operational rules defining the labelled transition relation are those in Table 4 together with those defining the ORC semantics (see [29, Table 3]) extended to KORC configurations in standard way⁵. For example, the rule for the left component of symmetric parallel composition extends to configurations as follows:

$$\frac{(f, \mathcal{K}) \xrightarrow{a} (f', \mathcal{K}')}{(f \mid g, \mathcal{K}) \xrightarrow{a} (f' \mid g, \mathcal{K})}$$

Notably, site and expression calls cannot modify the set \mathcal{K} , only the KORC actions **out**, **eval**, **in** and **read** can.

The rules in Table 4, like those in the semantics of KLAIM, exploit two auxiliary functions: $\mathcal{E}[\![-]\!]_{\rho}$ for evaluating tuples/templates using the allocation environment ρ , and $match(\cdot, \cdot)$ for verifying the compliance of a tuple w.r.t. a template and associating basic values, locality names and processes to corresponding variables in templates.

Let us now comment on the rules in Table 4. All actions evolve to expression **0** (which has no observable transitions), act on a net named η , require the

⁵ Since KORC inherits pairwise disjoint variables sets from KLAIM, the definition of the ORC pattern-matching function $\mathcal{M}(\cdot, \cdot)$ has been revised to guarantee that each variable only matches with values of the corresponding category (see [29, Section 2.3]).

$\frac{\rho(\ell) = s' \quad \mathcal{E} \llbracket t \rrbracket_\rho = t' \quad (\text{fn}(t') \cup \{s'\}) \not\subseteq (\text{bn}(N) \cup \bar{s})}{\begin{array}{c} (\mathbf{out}(t) @ \eta : \ell, \mathcal{K} \uplus \{\eta ::_\rho (\nu \bar{s})(N \parallel s' ::_{\rho'} \mathbf{nil})\}) \\ \xrightarrow{! \langle \rangle} (\mathbf{0}, \mathcal{K} \uplus \{\eta ::_\rho (\nu \bar{s})(N \parallel s' ::_{\rho'} \langle t' \rangle)\}) \end{array}} \text{ (Korc-out)}$
$\frac{\rho(\ell) = s' \quad (\text{fn}(P) \cup \{s'\}) \not\subseteq (\text{bn}(N) \cup \bar{s})}{\begin{array}{c} (\mathbf{eval}(P) @ \eta : \ell, \mathcal{K} \uplus \{\eta ::_\rho (\nu \bar{s})(N \parallel s' ::_{\rho'} \mathbf{nil})\}) \\ \xrightarrow{! \langle \rangle} (\mathbf{0}, \mathcal{K} \uplus \{\eta ::_\rho (\nu \bar{s})(N \parallel s' ::_{\rho'} P)\}) \end{array}} \text{ (Korc-eval)}$
$\frac{\rho(\ell) = s' \quad \text{match}(\mathcal{E} \llbracket T \rrbracket_\rho, t) = \sigma \quad (\text{fn}(T) \cup \{s'\}) \not\subseteq (\text{bn}(N) \cup \bar{s})}{\begin{array}{c} (\mathbf{in}(T) @ \eta : \ell, \mathcal{K} \uplus \{\eta ::_\rho (\nu \bar{s})(N \parallel s' ::_{\rho'} \langle t \rangle)\}) \\ \xrightarrow{! \langle t \rangle} (\mathbf{0}, \mathcal{K} \uplus \{\eta ::_\rho (\nu \bar{s} \setminus \text{fn}(t))(N \parallel s' ::_{\rho'} \mathbf{nil})\}) \end{array}} \text{ (Korc-in)}$
$\frac{\rho(\ell) = s' \quad \text{match}(\mathcal{E} \llbracket T \rrbracket_\rho, t) = \sigma \quad (\text{fn}(T) \cup \{s'\}) \not\subseteq (\text{bn}(N) \cup \bar{s})}{\begin{array}{c} (\mathbf{read}(T) @ \eta : \ell, \mathcal{K} \uplus \{\eta ::_\rho (\nu \bar{s})(N \parallel s' ::_{\rho'} \langle t \rangle)\}) \\ \xrightarrow{! \langle t \rangle} (\mathbf{0}, \mathcal{K} \uplus \{\eta ::_\rho (\nu \bar{s} \setminus \text{fn}(t))(N \parallel s' ::_{\rho'} \langle t \rangle)\}) \end{array}} \text{ (Korc-read)}$
$\frac{N \mapsto N'}{(f, \mathcal{K} \uplus \{\eta ::_\rho N\}) \xrightarrow{\tau} (f, \mathcal{K} \uplus \{\eta ::_\rho N'\})} \text{ (Korc-net)}$

Table 4. KORC operational semantics (additional rules)

existence of the target node s' (which must not be restricted in η) and exploit the environment ρ for evaluating their arguments. We abbreviate $(\nu s_1) \dots (\nu s_n)$ to $(\nu \bar{s})$ with $\bar{s} = \langle s_1, \dots, s_n \rangle$. Actions **out** and **eval**, rules *(Korc-out)* and *(Korc-eval)*, can be performed only if the components they intend to insert in s' (i.e. the tuple $\langle t' \rangle$ or the process P) do not contain locality names restricted in η . If such actions can be performed, a signal $\langle \rangle$ is published. It has been decided to emit a signal and not to perform a τ event, to use the signal in further sequential or asymmetric parallel compositions (see rules *(Seq1)* and *(Asym2)* in [29, Table 3]). Similarly, actions **in** and **read**, rules *(Korc-in)* and *(Korc-read)*, can be performed only if the template T does not contain locality names restricted in η , because a private name cannot be matched by any name used outside the net (private names cannot be ‘guessed’). Instead, these actions can be performed if the tuple t they intend to withdraw/read contains some locality names restricted in η ; in this case, the restriction of such names is removed. If a matching datum t exists in the target node, actions **in** and **read** can proceed and publish the withdrawn/read tuple $\langle t \rangle$. Notice that, to properly integrate **in** and **read** actions with the binding operators of ORC, in rules *(Korc-in)* and *(Korc-read)* the generated substitution σ is not applied and the complete withdrawn/read tuple is published. The values in the returned tuple can be then caught via pattern-matching through sequential or asymmetric parallel compositions. Finally, rule *(Korc-net)* says that KLAIM nets in \mathcal{K} can freely evolve w.r.t. the evolution of expression f .

As mentioned above, the execution of actions **in** and **read** in KORC does not yield a substitution, but simply the publication of the involved tuple. However, **in** and **read** actions *à la* KLAIM can be easily modelled in KORC: e.g., an expression **in**(5, ! x)@ $\eta : \ell . f$, where a substitution for x is applied to f , can be rendered in KORC as **in**(5, ! y)@ $\eta : \ell > \langle 5, x \rangle > f$, where y is a fresh local variable. Moreover, in KLAIM, locality names can be private, i.e. restricted with operator (νs) and their freshness can be guaranteed by a middleware supporting the execution of a KLAIM net. Instead, the loosely coupled nature of the service-oriented architecture underlying KORC makes it more difficult to guarantee names freshness over a global net consisting of many independent KLAIM subnets. Therefore, in KORC, when a private name is extracted from a KLAIM net, through an **in/read** action, the name becomes public (like in the *open* rule of π -calculus [30]).

It is worth noticing that KORC is not equipped with specific linguistic primitives for composing programs, which are indeed designed to be separately executed. However, KORC programs can be easily composed by resorting to the three ORC orchestration operators. More specifically, if two programs act on the same set \mathcal{K} of named KLAIM nets, their composition is the program consisting of the set \mathcal{K} and the expression obtained by applying the composition operator to the two expressions of the argument programs. As an example, consider the two KORC programs (f, \mathcal{K}) and (g, \mathcal{K}) , the program corresponding to their sequential composition is $(f > \bar{p} > g, \mathcal{K})$. If the two programs act on different sets of nets, the composition is done similarly, except that the two sets must be composed by means of an appropriate union operator that guarantees the well-formedness of the resulting KORC program.

3 Korc at work on an e-commerce case study

In this section, we illustrate an application of KORC to a simplified but realistic electronic marketplace scenario, where a number of on-line stores allow client applications to read data about items availability and to place orders. We assume that each store has an on-line portal and relies on many ‘realworld’ stores, each of which with its own warehouse. Specifically, here we consider a client application that aims at finding a store that has in stock a given quantity of a specific item, by concurrently accessing different stores, and placing an order to the first store found. For the sake of presentation, we shall consider a scenario consisting of only three on-line stores. The outlined scenario can be rendered in KORC by:

$$(f, \{\eta_{store1} ::_{\rho} N_1, \eta_{store2} ::_{\rho} N_2, \eta_{store3} ::_{\rho} N_3, \eta_{client} ::_{\rho} (s ::_{\{\mathbf{self} \rightarrow s\}} \mathbf{nil})\})$$

where ρ stands for $\{l \mapsto s\}$ and each net N_i has the following form

$$\begin{aligned} & s_1 ::_{\{\mathbf{self} \rightarrow s_1, l \mapsto s, l_{next} \mapsto s_2, l_{end} \mapsto s_e\}} \langle t_{i1}^1 \rangle \mid \dots \mid \langle t_{i1}^{k_i} \rangle \\ & \parallel s_2 ::_{\{\mathbf{self} \rightarrow s_2, l \mapsto s, l_{next} \mapsto s_3, l_{end} \mapsto s_e\}} \langle t_{i2}^1 \rangle \mid \dots \mid \langle t_{i2}^{w_i} \rangle \\ & \parallel \dots \parallel s_{m_i} ::_{\{\mathbf{self} \rightarrow s_{m_i}, l \mapsto s, l_{next} \mapsto s_e, l_{end} \mapsto s_e\}} \langle t_{im}^1 \rangle \mid \dots \mid \langle t_{im}^{r_i} \rangle \\ & \parallel s ::_{\{\mathbf{self} \rightarrow s, l_{start} \mapsto s_1\}} \mathbf{nil} \end{aligned}$$

In KORC, each store $store\ i$ consists of a site $M_{store\ i}$, representing the on-line portal to place orders to the store (see expression g below), and a named net $\eta_{store\ i} ::_{\rho} N_i$ whose nodes s_j represent the data storages of its warehouses while node s is used for computational support. Each tuple t_{ij}^u represents the data of a specific item stored inside the warehouse s_j of the store $store\ i$. Specifically, such tuples have the form $\langle id, q, p \rangle$, where id is the item identifier, q (with $q > 0$) is the quantity available at the warehouse, and p is the price (which can be different from a warehouse to another). We assume that each node contains at most one tuple for each item identifier. Finally, the client application is rendered in KORC as the expression f and the net η_{client} . The latter contains a node s , initially empty, to elaborate the retrieved data. The expression f is defined as follows:

$$\begin{aligned} & \mathbf{eval}(\mathbf{FindItem}(\text{"itemId3"}, 20, \text{"reqId12"})) @ \eta_{store1} : l \\ & \mid \mathbf{eval}(\mathbf{FindItem}(\text{"itemId3"}, 20, \text{"reqId12"})) @ \eta_{store2} : l \\ & \mid \mathbf{eval}(\mathbf{FindItem}(\text{"itemId3"}, 20, \text{"reqId12"})) @ \eta_{store3} : l \\ & \mid f_{moveFromStore1} \mid f_{moveFromStore2} \mid f_{moveFromStore3} \mid g \end{aligned}$$

Basically, it represents a client's search request for 30 items⁶ of type "itemId3" whose maximum price per item that the client is willing to pay is less or equal to 20. To avoid that data of different search requests are erroneously mixed together, a request identifier, say "reqId12", is provided by the client and inserted into each tuple. Of course, the above expression could be parameterized w.r.t. item identifier, price, request identifier and quantity, but we prefer to leave it as it is for the sake of presentation.

Specifically, by means of three **eval** actions, the client expression spawns three copies of the process $\mathbf{FindItem}$ into the locality s of each store net. Such process looks for tuples having as arguments the item identifier "itemId3" and a price less or equal to 20. A copy of each tuple (extended with the request identifier "reqId12") that meets this requirement is stored in the locality s of the net. Then, by means of three expression calls $f_{moveFromStore\ i}$, as tuples are inserted into the node s of each store's net, they are moved to the node s of the client's net. Each expression $f_{moveFromStore\ i}$ is defined as a recursive expression performing a sequence of **in** and **out** actions:

$$\begin{aligned} f_{moveFromStore\ i} & \triangleq \mathbf{in}(\text{"itemId3"}, !x_q, !x_p, \text{"reqId12"}) @ \eta_{store\ i} : l \\ & > \langle \text{"itemId3"}, x_q, x_p, \text{"reqId12"} \rangle > \\ & \mathbf{out}(\text{"store\ i"}, \text{"reqId12"}, x_q) @ \eta_{client} : l \gg f_{moveFromStore\ i} \end{aligned}$$

where $f_1 \gg f_2$ is used as short-hand for $f_1 > \langle \rangle > f_2$. Notably, in performing such movements, information about prices and item identifiers are left out, while information about the source stores are added.

The KLAIM process $\mathbf{FindItem}$ is defined as follows:

$$\begin{aligned} \mathbf{FindItem}(\text{itemId}, \text{maxPrice}, \text{reqId}) & \triangleq \\ & \mathbf{eval}(\mathbf{Find}(\text{itemId}, \text{maxPrice}, \text{reqId})) @ l_{start} \end{aligned}$$

⁶ As it will be clearer later, the check of the availability of 30 items is performed by the subexpression g of f (to be more precise, by the three components g_i of g).

It simply activates a mobile process *Find* (i.e. an *agent*) in the ‘start’ locality of the hosting net. This mobile process, defined below, will visit all nodes of the net to find the availability of the wanted item:

$$\begin{aligned} Find(itemId, maxPrice, reqId) &\triangleq \\ &\mathbf{read}(itemId, !q, !p)@self. \\ &\quad \mathbf{if} (p \leq maxPrice) \mathbf{then out}(itemId, q, p, reqId)@l \\ &\quad | \mathbf{if} (l_{next} \neq l_{end}) \mathbf{then eval}(Find(itemId, maxPrice, reqId))@l_{next} \end{aligned}$$

The process simply checks if a tuple for the given item is present locally; if the price per item is not greater than the maximum price then it adds a corresponding tuple to the node *s* of the hosting net (referred by means of the locality variable *l*). Moreover, if there exists a next node to be visited, then a new copy of the process is spawned on such node. This second check exploits the locality variables *l_{next}* and *l_{end}*, which are properly bound by the allocation environment of each net node. For the sake of simplicity, in defining the above agent we have used a conditional construct (which can be easily programmed by exploiting the dynamic creation of new nodes and the parallel composition operator) and we have omitted trailing occurrences of **nil**.

The expression *g* is defined as follows:

$$\begin{aligned} g &\triangleq \mathbf{out}("sum", "store1", "reqId12", 0)@_{\eta_{client}} : l \\ &\quad | \mathbf{out}("sum", "store2", "reqId12", 0)@_{\eta_{client}} : l \\ &\quad | \mathbf{out}("sum", "store3", "reqId12", 0)@_{\eta_{client}} : l \\ &\quad | ((\mathbf{if}(x = "store1ok") \gg M_{store1}("itemId3", 30)) \\ &\quad \quad | (\mathbf{if}(x = "store2ok") \gg M_{store2}("itemId3", 30)) \\ &\quad \quad | (\mathbf{if}(x = "store3ok") \gg M_{store3}("itemId3", 30))) \\ &\quad < \langle x \rangle < (g_1 \mid g_2 \mid g_3) \end{aligned}$$

It adds to the node *s* of the client’s net three tuples containing the partial sum of the quantity of the requested item available at each store (initially set to 0). It also starts the concurrent evaluation of three expressions *g_i*, each of which computes the sum of the item quantity for a store and publishes the string “store *i* ok” if the store has in stock at least 30 items of the requested type. The asymmetric parallel composition operator is used here to bind the variable *x* with the (first published) string “store *i* ok” and to terminate the evaluation of the other functions *g_j*, with *j* ≠ *i*. Then, according to the published string, the corresponding site *M_{store i}* is called to place an order. We have exploited here the fundamental⁷ ORC site *if(b)*, which returns a signal $\langle \rangle$ if *b* evaluates to *true*, otherwise it does not respond.

Finally, an expression *g_i* is defined as follows:

$$\begin{aligned} g_i &\triangleq \mathbf{in}("store i", "reqId12", !y_q)@_{\eta_{client}} : l \\ &\quad > \langle "store i", "reqId12", y_q \rangle > \\ &\quad \mathbf{in}("sum", "store i", "reqId12", !y_{sum})@_{\eta_{client}} : l \end{aligned}$$

⁷ To effective programming in ORC, the language is equipped with a few ‘fundamental’ sites (e.g. *if(b)*, *let(x, y, ...)*) that have to be considered local and whose behavior is predefined and predictable [40].

$$\begin{aligned}
& > \langle \text{"sum"}, \text{"store } i", \text{"reqId12"}, y_{sum} \rangle > \\
& ((\text{if}(y_q + y_{sum} \geq 30) \gg \text{let}(\text{"store } i \text{ ok"})) \\
& \quad | (\text{if}(y_q + y_{sum} < 30) \gg \\
& \quad \quad (\text{out}(\text{"sum"}, \text{"store } i", \text{"reqId12"}, y_q + y_{sum}) @ \eta_{client} : l \gg g_i)))
\end{aligned}$$

Basically, this is a recursive expression that, at each recursive call, consumes a tuple containing an item availability and a tuple containing the actual sum, computes the sum between the read values and, if the sum is less than the desired number (i.e. 30) it produces a new ‘sum’ tuple and calls itself, otherwise publishes the string “store i ok” and terminates. Notably, to publish the string “store i ok”, expression g_i exploits the fundamental ORC site $\text{let}(x, y, \dots)$, which returns the argument values as a tuple.

4 Implementation issues

In this section, we first provide a brief overview of the implementations of the programming languages derived from ORC and KLAIM, then we give a glimpse of the proof-of-concept implementation of KORC.

Although ORC was originally conceived as a process calculus, it has then evolved into a complete language for programming orchestration-based concurrent applications [26]. Such a programming language provides the ORC’s orchestration operators and the site call construct with their original syntax, while expression definitions take the form $\text{def } f(x_1, \dots, x_n) = f_{body}$. The language is also equipped with arithmetic and logical operators, data structures, a conditional construct, and a variable binder construct **val** (e.g., **val** $x = 5$ binds x to 5). Moreover, Java classes can be accessed by an ORC expression as sites. To make a class available to an expression, a **site** declaration and a variable binding must be used like in the following example

```

site orcNode = com.orcNode
val client = orcNode(...)

```

where the variable **client** can be then used for calling functionalities provided by the Java class **com.orcNode**. To be accessed as an ORC site, a Java class must extend one of the specific classes provided with the ORC’s libraries (e.g. **EvalSite**). We refer the interested reader to [1] for a complete account of the ORC programming language and its supporting libraries⁸.

Similarly to ORC, also the process calculus KLAIM has been extended with high-level features, such as variable declarations, assignments, and (standard) control flow constructs, to effectively program distributed networked applications. The implementation of the resulting programming language, called X-KLAIM (eXtended KLAIM [5]), is based on a compiler, which generates Java code, and on the Java library KLAVA [4], which provides the run-time support for X-KLAIM actions within the generated code⁹. The KLAIM net N_1 belonging

⁸ We consider here the version 1.1.0 of the ORC’s implementation, whose source code and binaries can be downloaded from <http://orc.csres.utexas.edu>.

⁹ Complete documentation of X-KLAIM and KLAVA, together with source and binary files can be found at <http://music.dsi.unifi.it/klaim.html>.

to *store1* of the e-commerce case study introduced in Section 3 can be rendered in X-KLAIM as follows:

```
nodes
  shop11::{nextl ~ localhost:11002, endl ~ localhost:11005, l ~ localhost:11004}
    port 11001
    begin
      out("itemId1",10,11)@self; ...
    end;

  shop12::{...}
    port 11002 ...
  ...
endnodes
```

A net, as expected, is a collection of node definitions, which must be included within **nodes** and **endnodes**. A node, e.g. the first one in the net above, is defined by specifying its name (**shop11**), its allocation environment (containing, e.g., the mapping from the locality variable **nextl** to the locality **localhost : 11002**), the port (11001) where it is listening, and a set of processes running on it (**out("itemId1", 10, 11)@self; ...**). It is worth noticing that the (physical) locality of a node is not defined by its name, but by the IP address of the computer where the node will run (in our example, this always is **localhost**) together with its port number. Instead, as an example of process definition, consider the process *FindItem* exploited in the e-commerce case study:

```
rec FindItem [itemId:str, maxPrice:int, reqId:str]
  declare locname startl
  begin
    eval(Find(itemId,maxPrice,reqId))@startl
  end
```

To speed up the experimentation with the programming paradigm fostered by KORC, we have exploited the compile- and run-time support tools for ORC and KLAIM presented above to implement KORC. The underlying idea is the following: KORC expressions are rendered as standard ORC expressions that rely on ad-hoc sites for performing the KLAIM actions. Specifically, we have developed a Java class **com.orcNode**, extending **EvalSite**, that can be used to define a new type of ORC site and that relies on the KLAIVA library for performing the KLAIM actions. Since KLAIVA uses types for values different from those of ORC, e.g. **KString**, **KInteger**, etc., and allows patterns to use both actual and formal parameters, we have also developed another kind of ORC site, **com.orcTuple**, that can be used to create objects having the correct types for invoking the KLAIVA methods.

As an example of how a KORC expression is rendered in our implementation, the expression *f* of the e-commerce case study is written as follows

```
site orcNode = com.orcNode
site orcTuple = com.orcTuple

val client = orcNode("client",15000,"localhost",9999)
val store1 = orcNode("store1",15001,"localhost",9998)
val store2 = orcNode("store2",15002,"localhost",9997)
val store3 = orcNode("store3",15003,"localhost",9996)
val c = orcTuple()
```

```

def addLocality() =
  client.addEnv("lClient",14000) >> store1.addEnv("l",11004) >> ...

def moveFromStore1() =
  store1.in(c.tuple("id3",c.intFormal(),c.intFormal(),"reqId12"),c.locality("l"))
  > x > c.get(x,1) > z >
  client.out(c.tuple("store1","reqId12",z),c.locality("lClient")) >> moveFromStore1()

def ...

addLocality()
>> ( startSearch() >> (moveFromStore1() | moveFromStore2() | moveFromStore3() | g()) )

```

At the beginning, our sites `com.orcNode` and `com.orcTuple` are declared and assigned to some variables. Each `com.orcNode` site permits interfacing with a KLAIM net; thus, the corresponding variable can play the role of net name in the subsequent KORC actions. For example, the action `out("store1", "reqId12", z)@ η_{client} :l` is rendered as `client.out(c.tuple("store1", "reqId12", z), c.locality("l"))`, where `client` represents η_{client} .

It is worth noticing that a `com.orcNode` site corresponds to a node belonging to the corresponding KLAIM net (in the example above, for the `client` net such node has name `client` and locality `localhost : 9999`). Thus, specific methods have been provided to set the allocation environment of such nodes and to load processes into them: `addEnv` and `loadProcess`, respectively. Notice also that formal parameters are unnamed in `com.orcTuple` tuples and, hence, a `get` method has to be used after **in/read** actions to extract the values associated to the formal parameters by pattern-matching.

We refer the interested reader to [29] for the Java code of classes `com.orcNode` and `com.orcTuple`. Such classes can be downloaded from <http://cse.lab.imtlucca.it/korc/> and can be installed in ORC as any other Java class defining an external site. The KORC implementation has been tested with ORC 1.1.0, X-KLAIM 2.b9 and KLAIVA 2.b1. Due to lack of space, also the complete specification of the e-commerce case study, written in the syntax accepted by our tool, is relegated to [29].

5 Concluding Remarks

We have introduced KORC, a formalism aiming at reconciling the orchestration paradigm of ORC with the coordination one of KLAIM. Specifically, we have formally defined syntax and operational semantics of KORC, and we have developed a prototype implementation supporting KORC programming.

As witnessed by the case study presented in Section 3, the combined approach that we propose is very convenient to program distributed networked applications. In fact, on the one hand, the KLAIM approach alone does not permit exploiting the powerful ORC's orchestration operators and interacting with external sites. On the other hand, the ORC approach used alone is not suitable for distributed coordination tasks. This would require the use of dummy sites and would make programming complex and awkward.

In particular, while the operators for sequential composition and for symmetric parallel composition could be rendered in KLAIM by properly exploiting

action prefixing and parallel composition, it would be tricky to express ORC asymmetric parallel composition $\cdot < \bar{p} < \cdot$ in terms of KLAIM constructs. Indeed, $f < \bar{p} < g$ permits immediately terminating the evaluation of g when a given event occurs (i.e. g publishes a tuple) while KLAIM lacks primitives for interrupting processes. In general, as seen in the case study, asymmetric parallel composition is very suitable for orchestration purposes, e.g. to implement transactional behaviours and fault handling. Another relevant aspect where KORC improves on KLAIM concerns the capability of interacting with external ORC sites, which may act as proxies for different kinds of services and applications. This enables the possibility of contacting and, hence, coordinating web services.

Some of the drawbacks of relying only on ORC approach are evident from our case study. There, the data storages of the warehouses of a store are rendered in a natural way as nodes of a KLAIM net. In this way, in KORC, to check the availability of items of type “*itemId3*”, it is sufficient to perform the action **read**(“*itemId3*”, !*q*, !*p*)@**self** on the nodes; among all information stored in the tuple spaces about different kinds of items, by exploiting the pattern-matching mechanism, this action directly accesses the information for “*itemId3*”. If we would use ORC alone to model this aspect, we would have to create a site, for each data storage, that publishes all items available at the corresponding warehouse and, then, use the pattern-matching provided by sequential composition to identify “*itemId3*” among all published values. Another solution would be to implement the search completely at site-side, thus leaving just site calls at expression-side; the programmer would then be forced to use another language (i.e. Java) to complete the implementation of the case study rather than simply using ORC. Notice also that, unless a single site would handle the data of all warehouses (which would not be reasonable in a distributed setting), the ORC program has to contact separately all warehouse sites and then to elaborate the retrieved information. In KORC, all the data storages associated to a given store can be visited through a single mobile process.

Related work. From the theoretical point of view, the formalisms closest to ours are ORC and KLAIM. In fact, to define KORC, we have chosen them as representative of the broader classes of orchestration calculi (as, e.g., [28, 10, 23, 11, 7]) and coordination calculi for network-aware and mobility programming (as, e.g., [21, 24, 12, 39]). Relatively to these calculi, KORC does not provide new primitives, but it permits experimenting and reasoning on a novel programming paradigm combining orchestration and coordination operators.

In the web services literature [34], the terms *orchestration* and *choreography* are used to describe composition of web services. Orchestration describes how services can interact from the perspective of one party (*local descriptions*), while choreography tells of the sequence of messages according to a global perspective, where each party describes the part that plays in the choreography (*global descriptions*). Means to check conformance of local and global descriptions have been defined in [9, 10, 27], by relying on bisimulation-like relations, and in [25], by relying on session types. In KORC, the ORC part describes the orchestration, while the KLAIM part represents a form of collaborative coordination that can

be used to enforce the involved parties to adhere to a given protocol, which can be thought of as a sort of choreography. Notably, both components of a KORC program play an active role, i.e. represent running programs, and describe different parts of the same system. This makes our approach different from the above mentioned works, where a choreography is intended to be either checked for conformance w.r.t. an orchestration of the different parties, or projected onto individual parties; in both cases, only orchestration is actually executed.

From the technological point of view, our work falls within the line of research that aims at developing programming frameworks based on process calculi. Among the several proposals, we want to mention below those designed for programming distributed networked applications. JCaSPiS [3] is a Java implementation of the service-oriented calculus CaSPiS [7] that, as well as KORC, takes inspiration by ORC (in particular, for the use of the sequential composition operator, called *pipeline*, over value streams). CaSPiS's implementation is based on the generic Java framework IMC [2] that provides recurrent mechanisms for network applications and, hence, can be used as a middleware for the implementation of different process calculi. JOLIE [32] is an interpreter written in Java for a programming language based on the process calculus SOCK [23], which is a formalism inspired by the WS-BPEL language for formalizing some fundamental concepts of Service-Oriented Computing, such as the design of a service behaviour, its deployment, and the composition of services within a system. JSCL [20] is a Java-based coordination middleware for services based on the event notification paradigm of the Signal Calculus [19], a variant of the π -calculus with explicit primitives to deal with event notification and component distribution. Finally, PiDuce [13] is a distributed run-time environment that implements a variant of the asynchronous π -calculus extended with native XML values, datatypes and patterns. The environment also permits interacting and experimenting with web services technologies.

Ongoing and future work. At foundational level, we intend to investigate the extension of KORC with name passing communication. Indeed, the ORC's formalization considered in this paper, drawn from [40], does not allow expressions to receive site names and use them in site calls, e.g. the term $M() > x > x(5)$ is not an ORC expression since the variable x cannot occur as a site name in the call $x(5)$. However, in other formalizations of ORC, see e.g. [31], sites are intended to be published as values by other sites and then called or used as parameters. Moreover, in KORC, besides site name passing, also net name passing is disallowed. In fact, a language for programming networked applications that permits passing net names but not site names would not be particularly meaningful.

We intend also to revise the programming language based on KORC presented in Section 4 to make it more usable by programmers. For example, KLAIM actions should have a syntax more similar to that shown in Table 3 and permit the direct use of named formal parameters. This could be realized, e.g., by means of a pre-compiling step. To further simplifying the task of writing KORC programs, we also intend to provide programmers with an Eclipse-based development en-

environment relying on the Xtext framework [18]. Finally, while KORC is basically an extension of ORC with KLAIM actions and nets, we are also currently investigating a sort of reverse extension, i.e. KLAIM with mechanisms for calling sites (specifically, web services via SOAP over HTTP). Such extension mainly involves the KLAIM middleware (i.e. X-KLAIM and KLAVA) rather than the process calculus itself, since we would still rely on standard **out/in** actions for interacting with web services.

References

1. Orc Reference Manual. Technical report, University of Texas at Austin, 2011. Available at <http://orc.csres.utexas.edu/documentation.shtml>.
2. L. Bettini, R. De Nicola, D. Falassi, M. Lacoste, and M. Loreti. A Flexible and Modular Framework for Implementing Infrastructures for Global Computing. In *DAIS, LNCS* 3543, pp. 181–193. Springer, 2005.
3. L. Bettini, R. De Nicola, M. Lacoste, and M. Loreti. Implementing Session Centered Calculi. In *COORDINATION, LNCS* 5052, pp. 17–32. Springer, 2008.
4. L. Bettini, R. De Nicola, and R. Pugliese. Klava: a Java Package for Distributed and Mobile Applications. *Software - Practice and Experience*, 32(14):1365–1394, 2002.
5. L. Bettini, R. De Nicola, and R. Pugliese. X-Klaim and Klava: Programming Mobile Code. In *TOSCA 2001, ENTCS* 62. Elsevier, 2001.
6. L. Bettini et al. The Klaim Project: Theory and Practice. In *Global Computing, LNCS* 2874, pp. 88–150. Springer, 2003.
7. M. Boreale, R. Bruni, R. De Nicola, and M. Loreti. Sessions and Pipelines for Structured Service Programming. In *FMOODS, LNCS* 5051, pp. 19–38. Springer, 2008.
8. M. Buscemi and U. Montanari. CC-Pi: A Constraint-Based Language for Specifying Service Level Agreements. In *ESOP, LNCS* 4421, pp. 18–32. Springer, 2007.
9. N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro. Choreography and Orchestration: A Synergic Approach for System Design. In *ICSOC, LNCS* 3826, pp. 228–240. Springer, 2005.
10. N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro. Choreography and orchestration conformance for system design. In *COORDINATION, LNCS* 4038, pp. 63–81. Springer, 2006.
11. M. Carbone, K. Honda, and N. Yoshida. Structured Communication-Centred Programming for Web Services. In *ESOP, LNCS* 4421, pp. 2–17. Springer, 2007.
12. L. Cardelli and A.D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177213, 2000.
13. S. Carpineti, C. Laneve, and L. Padovani. PiDuce - a project for experimenting Web services technologies. *Science of Comput. Program.*, 74(10):777–811, 2009.
14. P. Ciancarini and D. Rossi. Jada - Coordination and Communication for Java Agents. In *Mobile Object Systems, LNCS* 1222, pp. 213–226. Springer, 1997.
15. R. De Nicola, G. Ferrari, and R. Pugliese. KLAIM: A Kernel Language for Agents Interaction and Mobility. *Trans. on Software Engineering*, 24(5):315–330, 1998.
16. R. De Nicola, D. Gorla, and R. Pugliese. On the expressive power of KLAIM-based calculi. *Theor. Comput. Sci.*, 356(3):387–421, 2006.
17. E. Denti, A. Natali, and A. Omicini. On the expressive power of a language for programming coordination media. In *SAC*, pp. 169–177. ACM, 1998.

18. H. Behrens et al. Xtext 1.0, 2010. <http://www.eclipse.org/xtext>.
19. G. Ferrari, R. Guanciale, and D. Strollo. Event based service coordination over dynamic and heterogeneous networks. In *ICSOC, LNCS 4294*, pp. 453–458. Springer, 2006.
20. G. Ferrari, R. Guanciale, and D. Strollo. JSCL: A middleware for service coordination. In *FORTE, LNCS 4229*, pp. 46–60. Springer, 2006.
21. C. Fournet, G. Gonthier, J.J. Levy, L. Maranget, and D. Remy. A Calculus of Mobile Agents. In *CONCUR, LNCS 1119*, pp. 406–421. Springer, 1996.
22. D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.
23. C. Guidi, R. Lucchi, R. Gorrieri, N. Busi, and G. Zavattaro. SOCK: A Calculus for Service Oriented Computing. In *ICSOC, LNCS 4294*, pp. 327–338. Springer, 2006.
24. M. Hennessy and J. Riely. Resource access control in systems of mobile agents. *Information and Computation*, 173(1):82–120, 2002.
25. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL*, pp. 273–284. ACM Press, 2008.
26. D. Kitchin, A. Quark, W.R. Cook, and J. Misra. The Orc Programming Language. In *FMOODS/FORTE, LNCS 5522*, pp. 1–25. Springer, 2009.
27. I. Lanese, C. Guidi, F. Montesi, and G. Zavattaro. Bridging the Gap between Interaction- and Process-Oriented Choreographies. In *SEFM*, pp. 323–332. IEEE, 2008.
28. A. Lapadula, R. Pugliese, and F. Tiezzi. A Calculus for Orchestration of Web Services. In *ESOP, LNCS 4421*, pp. 33–47. Springer, 2007.
29. A. Margheri, R. De Nicola, and F. Tiezzi. Orchestrating Tuple-based Languages (full version). Technical report, IMT Advanced Studies Lucca, 2011. Available at <http://cse.lab.imtlucca.it/korc/>.
30. R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, I and II. *Information and Computation*, 100(1):1–40, 41–77, 1992.
31. J. Misra and W.R. Cook. Computation Orchestration: A Basis for Wide-Area Computing. *Journal of Software and Systems Modeling*, 6(1):83–110, 2007.
32. F. Montesi, C. Guidi, R. Lucchi, and G. Zavattaro. JOLIE: a Java Orchestration Language Interpreter Engine. In *MTCoord, ENTCS 181*, pp. 19–33. Elsevier, 2007.
33. OASIS WSBPEL TC. Web Services Business Process Execution Language Version 2.0. Technical report, OASIS, April 2007.
34. C. Peltz. Web Services Orchestration and Choreography. *Computer*, 36(10):46–52, 2003.
35. G.P. Picco, A.L. Murphy, and G. Roman. Lime: Linda meets mobility. In *ICSE*, pp. 368–377. ACM, 1999.
36. A.I.T. Rowstron. WCL: A Co-ordination Language for Geographically Distributed Agents. *World Wide Web*, 1(3):167–179, 1998.
37. R. Tolksdorf. Laura: A Coordination Language for Open Distributed Systems. In *ICDCS*, pp. 39–46. IEEE, 1993.
38. W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.
39. M. Wand and I. Siveroni. Constraint systems for useless variable elimination. In *POPL*, pp. 291–302. ACM, 1999.
40. I. Wehrman, D. Kitchin, W.R. Cook, and J. Misra. A timed semantics of orc. *Theoretical Computer Science*, 402(2-3):234–248, 2008.