Orchestrating Tuple-based Languages*

Rocco De Nicola[1,2], Andrea Margheri[1], and Francesco Tiezzi[2]

[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 [31], 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., [26, 8, 10, 21, 11, 7]), we will focus on Orc [29, 35], 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.

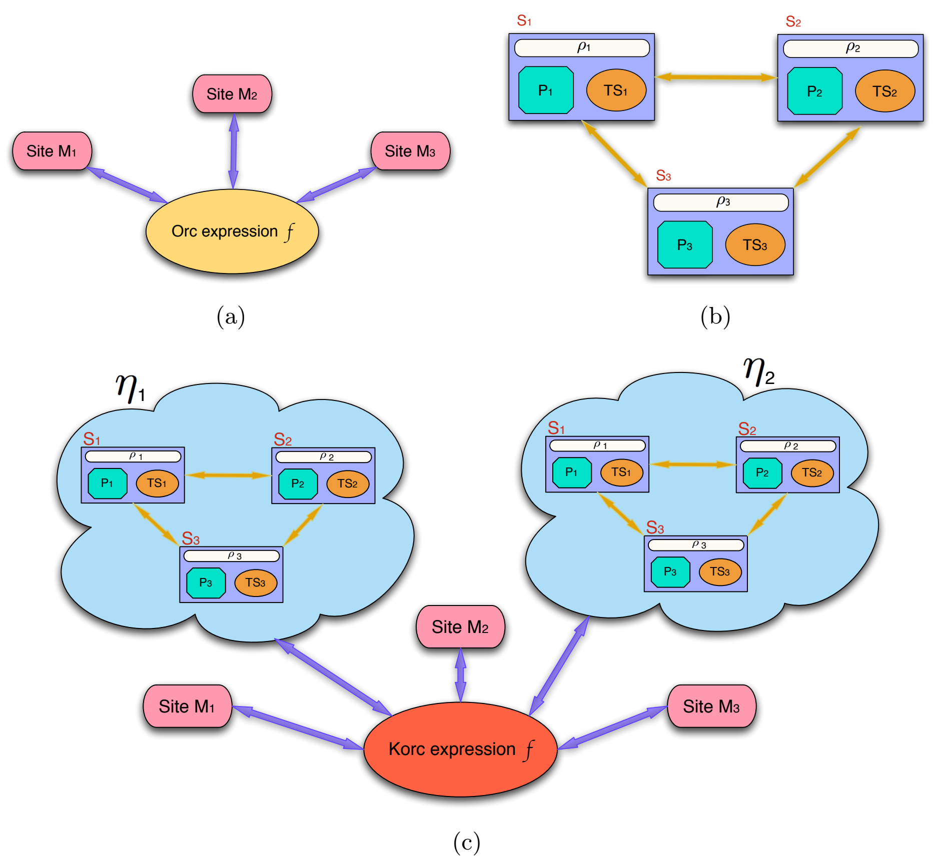

Fig. 1. The ORC (a), KLAIM (b) and KORC (c) approaches

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.

Although 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 [33]), 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, here, we would like to focus on KLAIM [14, 6, 15], a coordination language specifically designed to program distributed systems consisting of mobile components interacting through multiple distributed tuple spaces. KLAIM's communication model builds over, and extends, Linda's notion of generative communication through a single shared tuple space [20] and its primitives allow programmers to

$$\begin{array}{llll}
\text{(Expressions)} & f, g & ::= & M(\bar{p}) \quad | \quad E(\bar{p}) \quad | \quad f > \bar{p} > g \quad | \quad f \mid g \quad | \quad f < \bar{p} < g \\
\text{(Parameters)} & p & ::= & x \quad | \quad m
\end{array}$$

Table 1. Orc syntax

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 [35] and [15], 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 coordination language based on tuple spaces, we consider the polyadic variant of ORC informally described in [24] that permits using tuples as parameters rather than single values.

The overbar $^-$ over a name denotes tuples of parameters, thus $\bar{m}$ is the compact notation for the tuple of values $\langle m_1, \ldots, m_n \rangle$ (with $n \geq 0$). Variables in the same tuple are pairwise distinct. The empty tuple, written $\langle\rangle$, corresponds to a *signal*, the ORC unit value that has no additional information.

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 the elementary expressions $M(\bar{p})$, i.e. site calls, and $E(\bar{p})$, i.e. expression calls. The variables within $\bar{p}$ are *bound* in g for the expressions $f > \bar{p} > g$ and $g < \bar{p} < f$. We use $\mathrm{fv}(f)$ 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 only the variables $\bar{x}$ are free in f, i.e. $\bar{x} = \mathrm{fv}(f)$. The evaluation of an expressions may call a number of sites and returns a (possibly empty) stream of (tuple of) values.

An ORC expression can be either a site call, or an expression call or a composition of expressions according to one of the three basic orchestration patterns.

Site call: a site call can have the form $M(\bar{p})$, where the site name 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 $\bar{p}$ (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[3].

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 $\bar{m}$).

[3] The tuples published by f are consumed within $f > \bar{p} > g$.

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

Table 2. Klaim syntax

The operational semantics of Orc is given in terms of a labelled transition relation and an auxiliary function for pattern-matching on semi-structured data. Due to space limitations, we refer the interested reader to [27] for a full account of the Orc's 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',... range over *locality names* (i.e. network addresses); **self**, l, l',... range over *locality variables* (i.e. aliases for addresses); ℓ, ℓ',... range over locality names and variables; x, y,... range over *value variables*; X, Y,... range over *process variables*; e, e',... range over *expressions*[4]; A, B,... range over *process identifiers*[5]. 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. evaluated tuples and processes) can be allocated. A *node* is a triple $s ::_\rho C$, where locality s is the address of the node, ρ is the allocation environment and C are the host 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. 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.

[4] The precise syntax of expressions is deliberately not specified, but we assume that they contain, at least, basic values (ranged over by v, v',...) and variables.

[5] We assume that each process identifier of the form $A(f_1, \ldots, f_n) \triangleq P$, where f_i are pairwise distinct, has a unique definition, visible from any locality. Like for Orc, $\bar{p}$ and $\bar{f}$ denote tuples of actual and formal parameters, respectively.

Processes are the KLAIM active computational units that are built up from the special process **nil**, that does not perform any action, and from the basic operations by means of action prefixing $\alpha.P$, parallel composition $P_1 \mid P_2$ and process definition. Process may be executed concurrently either at the same locality or at different localities and can perform five different basic operations, called 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 creation of 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. **out** and **eval** are non-blocking actions and implement *static* and *dynamic scoping* disciplines, respectively (see [15, 27]).

Names and variables occurring in KLAIM processes and nets can be *bound*. More precisely, prefix $\mathbf{newloc}(s).P$ binds name s in P, and, similarly, net restriction $(\nu s)N$ binds s in N. The sets $fn(\cdot)$ and $bn(\cdot)$ of, respectively, free and bound locality names of a term are defined accordingly. Prefixes $\mathbf{in}(\ldots, !\_, \ldots)@\ell.P$ and $\mathbf{read}(\ldots, !\_, \ldots)@\ell.P$ 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 [27] 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

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

Table 3. KORC syntax

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 [15] and [27, 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 $\longmapsto$ over KLAIM nets (see [27, Table 7]). As in the semantics of ORC, label a is generated by the following grammar:

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

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 [27, Table 3]) extended to KORC configurations in standard way[6]. 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, expolit 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

[6] Since KORC inherits pairwise disjoint variables sets from KLAIM, the definition of the pattern-matching function $\mathcal{M}(\cdot, \cdot)$ has been revised to guarantee that each variable only matches with values of the corresponding category (see [27, Section 2.3]).

$$\frac{\rho(\ell) = s' \qquad \mathcal{E}[\![\, t \,]\!]_\rho = t' \qquad (fn(t') \cup \{s'\}) \not\subseteq (bn(N) \cup \bar{s})}{\begin{array}{l}(\mathbf{out}(t)@\,\eta : \ell, \mathcal{K} \uplus \{\eta ::_\rho (\nu\bar{s})(N \parallel s' ::_{\rho'} \mathbf{nil})\}) \\ \quad \xrightarrow{!\langle\rangle} (\mathbf{0}, \mathcal{K} \uplus \{\eta ::_\rho (\nu\bar{s})(N \parallel s' ::_{\rho'} \langle t' \rangle)\})\end{array}} \textit{(Ko-out)}$$

$$\frac{\rho(\ell) = s' \qquad (fn(P) \cup \{s'\}) \not\subseteq (bn(N) \cup \bar{s})}{\begin{array}{l}(\mathbf{eval}(P)@\,\eta : \ell, \mathcal{K} \uplus \{\eta ::_\rho (\nu\bar{s})(N \parallel s' ::_{\rho'} \mathbf{nil})\}) \\ \quad \xrightarrow{!\langle\rangle} (\mathbf{0}, \mathcal{K} \uplus \{\eta ::_\rho (\nu\bar{s})(N \parallel s' ::_{\rho'} P)\})\end{array}} \textit{(Ko-eval)}$$

$$\frac{\rho(\ell) = s' \qquad match(\mathcal{E}[\![\, T \,]\!]_\rho, t) = \sigma \qquad (fn(T) \cup \{s'\}) \not\subseteq (bn(N) \cup \bar{s})}{\begin{array}{l}(\mathbf{in}(T)@\,\eta : \ell, \mathcal{K} \uplus \{\eta ::_\rho (\nu\bar{s})(N \parallel s' ::_{\rho'} \langle t \rangle)\}) \\ \quad \xrightarrow{!\langle t\rangle} (\mathbf{0}, \mathcal{K} \uplus \{\eta ::_\rho (\nu\, \bar{s} \backslash fn(t))(N \parallel s' ::_{\rho'} \mathbf{nil})\})\end{array}} \textit{(Ko-in)}$$

$$\frac{\rho(\ell) = s' \qquad match(\mathcal{E}[\![\, T \,]\!]_\rho, t) = \sigma \qquad (fn(T) \cup \{s'\}) \not\subseteq (bn(N) \cup \bar{s})}{\begin{array}{l}(\mathbf{read}(T)@\,\eta : \ell, \mathcal{K} \uplus \{\eta ::_\rho (\nu\bar{s})(N \parallel s' ::_{\rho'} \langle t \rangle)\}) \\ \quad \xrightarrow{!\langle t\rangle} (\mathbf{0}, \mathcal{K} \uplus \{\eta ::_\rho (\nu\, \bar{s} \backslash fn(t))(N \parallel s' ::_{\rho'} \langle t \rangle)\})\end{array}} \textit{(Ko-read)}$$

$$\frac{N \longmapsto N'}{(f, \mathcal{K} \uplus \{\eta ::_\rho N\}) \xrightarrow{\tau} (f, \mathcal{K} \uplus \{\eta ::_\rho N'\})} \textit{(Ko-net)}$$

Table 4. KORC operational semantics (additional rules)

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 $\mathbf{0}$ (which has no observable transitions), act on a net named η, require the existence of the target node s' (which must not be restricted in η) and exploit the allocation environment ρ for evaluating their arguments. We abbreviate $(\nu s_1)\ldots(\nu s_n)$ to $(\nu\bar{s})$ with $\bar{s} = \langle s_1, \ldots, s_n \rangle$. Actions **out** and **eval**, rules *(Ko-out)* and *(Ko-eval)*, can be performed only if the components they intend to insert in s' (i.e. the evaluated 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 [27, Table 3]). Similarly, actions **in** and **read**, rules *(Ko-in)* and *(Ko-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 *(Ko-*

in) and *(Ko-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 *(Ko-net)* says that KLAIM nets in $\mathcal{K}$ can freely evolve w.r.t. the evolution of expression f.

In KORC, execution of actions **in** and **read** does not yield a substitution, but simply the publication of the involved tuple. However, as mentioned above, **in** and **read** actions à la KLAIM can be easily modelled in KORC: expression $\mathbf{in}(5, !x)@\,\eta : \ell\,.f$, where a substitution for x is applied to f, can be rendered in KORC as $\mathbf{in}(5, !y)@\,\eta : \ell > \langle 5, x\rangle > f$, 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 [28].

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:

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

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

$$s_1\ ::_{\{\mathbf{self}\mapsto s_1, l\mapsto s, l_{next}\mapsto s_2, l_{end}\mapsto s_e\}}\ \langle t_{i1}^{1}\rangle \mid \ldots \mid \langle t_{i1}^{k_i}\rangle \mid \langle \text{“}next\text{”}, l_{next}\rangle$$

$$\begin{array}{l} \parallel\ s_2\ ::_{\{\mathbf{self}\mapsto s_2, l\mapsto s, l_{next}\mapsto s_3, l_{end}\mapsto s_e\}}\ \langle t^1_{i2}\rangle \mid \ldots \mid \langle t^{w_i}_{i2}\rangle \mid \langle \text{“}next\text{”}, l_{next}\rangle \\ \parallel\ \ldots\ \parallel\ s_{m_i}\ ::_{\{\mathbf{self}\mapsto s_{m_i}, l\mapsto s, l_{next}\mapsto s_e, l_{end}\mapsto s_e\}}\ \langle t^1_{im}\rangle \mid \ldots \mid \langle t^{r_i}_{im}\rangle \mid \langle \text{“}next\text{”}, l_{next}\rangle \\ \parallel\ s\ ::_{\{\mathbf{self}\mapsto s, l_1\mapsto s_1\}}\ \langle \text{“}start\text{”}, l_1\rangle \end{array}$$

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 computation support. Each tuple t^u_{ij} 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:

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

Basically, it represents a client's search request for 30 items[7] 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 $FindItem$ 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:

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

where $f_1 \gg f_2$ is used as short-hand for $f_1 > \langle\cdot\rangle > f_2$. 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 $FindItem$ is defined as follows:

[7] 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).

$$\begin{array}{l} FindItem(itemId, maxPrice, reqId) \;\triangleq \\ \mathbf{read}(\text{“}start\text{”}, !l_{start})@\mathbf{self}.\ \mathbf{eval}(Find(itemId, maxPrice, reqId))@l_{start} \end{array}$$

It locally reads the 'start' locality of the hosting net and then activates there a mobile process *Find* (i.e. an *agent*) that will visit all net nodes to find the availability of the wanted item. The mobile process is defined as follows:

$$\begin{array}{l} Find(itemId, maxPrice, reqId) \;\triangleq \\ \mathbf{read}(itemId, !q, !p)@\mathbf{self}. \\ \qquad \mathbf{if}\ (p \leqslant maxPrice)\ \mathbf{then}\ \mathbf{out}(itemId, q, p, reqId)@l \\ \mid \mathbf{read}(\text{“}next\text{”}, !l_n)@\mathbf{self}. \\ \qquad \mathbf{if}\ (l_n \neq l_{end})\ \mathbf{then}\ \mathbf{eval}(Find(itemId, maxPrice, reqId))@l_n \end{array}$$

The process simply checks if a tuple for the given item is present locally; if the item per price is not greater than the maximum price then it add a corresponding tuple to the node s of the hosting net (referred by means of the locality variable l). Moreover, it reads the locality of the next node and, if it there exists, than a new copy of the process is spawned on such node. This second check exploits the locality variable l_{end} that is 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:

$$\begin{array}{rl} g \;\triangleq & \mathbf{out}(\text{“}sum\text{”}, \text{“}store1\text{”}, \text{“}reqId12\text{”}, 0)@\,\eta_{client} : l \\ & \mid\ \mathbf{out}(\text{“}sum\text{”}, \text{“}store2\text{”}, \text{“}reqId12\text{”}, 0)@\,\eta_{client} : l \\ & \mid\ \mathbf{out}(\text{“}sum\text{”}, \text{“}store3\text{”}, \text{“}reqId12\text{”}, 0)@\,\eta_{client} : l \\ & \mid\ (\ (\ if(x = \text{“}store1ok\text{”}) \gg M_{store1}(\text{“}itemId3\text{”}, 30)\ ) \\ & \qquad \mid\ (\ if(x = \text{“}store2ok\text{”}) \gg M_{store2}(\text{“}itemId3\text{”}, 30)\ ) \\ & \qquad \mid\ (\ if(x = \text{“}store3ok\text{”}) \gg M_{store3}(\text{“}itemId3\text{”}, 30)\ )\ ) \\ & \quad < \langle x \rangle < (\ g_1 \mid g_2 \mid g_3\ ) \end{array}$$

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 string "*store i ok*" and to terminate the evaluation of the other functions g_j, with $j \neq i$. Then, according to the published string, the corresponding site $M_{store\,i}$ is called to place an order. We have exploited here the fundamental[8] Orc site $if(b)$, which returns a signal $\langle \cdot \rangle$ if b evaluates to *true*, otherwise it does not respond.

[8] To effective programming in Orc, the language is equipped with a few 'fundamental' sites (e.g. $if(b)$, $let(x, y, \ldots)$) that have to be considered local and whose behavior is predefined and predictable [35].

Finally, an expression g_i is defined as follows:

$$\begin{aligned}
g_i \triangleq\ & \mathbf{in}(\text{“}\mathit{store\,i}\text{”}, \text{“}\mathit{reqId}12\text{”}, !y_q)@\,\eta_{client} : l \\
& > \langle \text{“}\mathit{store\,i}\text{”}, \text{“}\mathit{reqId}12\text{”}, y_q \rangle > \\
& \mathbf{in}(\text{“}\mathit{sum}\text{”}, \text{“}\mathit{store\,i}\text{”}, \text{“}\mathit{reqId}12\text{”}, !y_{sum})@\,\eta_{client} : l \\
& > \langle \text{“}\mathit{sum}\text{”}, \text{“}\mathit{store\,i}\text{”}, \text{“}\mathit{reqId}12\text{”}, y_{sum} \rangle > \\
& (\ (\ \mathit{if}(y_q + y_{sum} \geqslant 30) \ \gg\ \mathit{let}(\text{“}\mathit{store\,i\,ok}\text{”})\ ) \\
& \quad |\ (\ \mathit{if}(y_q + y_{sum} < 30) \ \gg \\
& \qquad\qquad \mathbf{out}(\text{“}\mathit{sum}\text{”}, \text{“}\mathit{store\,i}\text{”}, \text{“}\mathit{reqId}12\text{”}, y_q + y_{sum})@\,\eta_{client} : l\ ) \ \gg\ g_i\ )
\end{aligned}$$

Basically, this 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 $let(x, y, \ldots)$, 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 [24]. 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 `def f(x1, ..., xn)` $= f_{body}$. 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 called 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 libraries[9].

Similarly to Orc, also the process calculus Klaim has been extended with high-level features, such as variable declarations, assignments, and (standard)

[9] The source code and binaries of the Orc's implementation can be downloaded from http://orc.csres.utexas.edu.

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 code. The KLAIM net N_1 belonging to *store*1 of the e-commerce case study introduced in Section 3 can be rendered in X-KLAIM as follows:

```
nodes
  shop11::{succ ~ localhost:11002,finish ~ localhost:11005, s ~ localhost:11004}
    port 11001
    begin
        out("next", succ)@self; out("id1",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 `succ` to the locality `localhost : 11002`), the port (`11001`) where it is listening, and a set of processes running on it (`out("next", succ)@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 var locstart: loc
  begin
       read("start",!locstart)@self; eval(Find(itemId,maxPrice,reqId))@locstart
  end
```

Notice that `str` and `int` are the standard base types for strings and integers, while `loc` is the type for locality variables[10].

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 KLAVA library for performing the KLAIM actions. Since KLAVA uses types for values different from those of ORC, i.e. `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 KLAVA methods.

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

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("l",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("l")) >> 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 $\mathbf{out}(\text{“}store1\text{”}, \text{“}reqId12\text{”}, z)@\eta_{client} : l$ 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 [27] for the Java code of classes `com.orcNode` and `com.orcTuple`. Such classes can be downloaded from http://rap.dsi.unifi.it/korc/korc.zip 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 KLAVA 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 [27].

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 "*itemId*3", it is sufficient to perform the action **read**("*itemId*3", $!q, !p$)@**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 "*itemId*3". 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 "*itemId*3" 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., [26, 10, 21, 11, 7]) and coordination calculi for network-aware and mobility programming (as, e.g., [19, 22, 12, 34]). 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 [32], 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, 25], by relying on bisimulation-like relations, and in [23], 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 (Calculus of Sessions and Pipelines, [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 (Implementing Mobile Calculi, [2]) that provides recurrent mechanisms for network applications and, hence, can be used as a middleware for the implementation of different process calculi. JOLIE [30] is an interpreter written in Java for a programming language based on the process calculus SOCK (Service Oriented Computing Kernel, [21]), 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 (Java Signal Core Layer, [18]) is a Java-based coordination middleware for services based on the event notification paradigm of the Signal Calculus [17], 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 [35], 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. [29], 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 also plan to investigate extension of Korc with other orchestration/coordination primitives, like the 'otherwise' operation introduced in [24].

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 environment relying on the Xtext framework [16]. 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. 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. *Softw. - Pract. and Exper.*, 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):177–213, 2000.
13. S. Carpineti, C. Laneve, and L. Padovani. PiDuce - a project for experimenting Web services technologies. *Sci. Comput. Program.*, 74(10):777–811, 2009.
14. 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.
15. R. De Nicola, D. Gorla, and R. Pugliese. On the expressive power of KLAIM-based calculi. *Theor. Comput. Sci.*, 356(3):387–421, 2006.
16. H. Behrens et al. Xtext 1.0, 2010. http://www.eclipse.org/Xtext/.
17. 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.
18. G. Ferrari, R. Guanciale, and D. Strollo. JSCL: A middleware for service coordination. In *FORTE*, *LNCS* 4229, pp. 46–60. Springer, 2006.
19. 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.
20. D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.
21. 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.
22. M. Hennessy and J. Riely. Resource access control in systems of mobile agents. *Information and Computation*, 173(1):82–120, 2002.
23. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *Proc. of 35th ACM Symposium on Principles of Programming Languages (POPL)*, pp. 273–284. ACM Press, 2008.
24. D. Kitchin, A. Quark, W.R. Cook, and J. Misra. The Orc Programming Language. In *FMOODS/FORTE*, *LNCS* 5522, pp. 1–25. Springer, 2009.
25. Ivan Lanese, Claudio Guidi, Fabrizio Montesi, and Gianluigi Zavattaro. Bridging the Gap between Interaction- and Process-Oriented Choreographies. In *SEFM*, pp. 323–332. IEEE Computer Society, 2008.
26. A. Lapadula, R. Pugliese, and F. Tiezzi. A Calculus for Orchestration of Web Services. In *ESOP*, *LNCS* 4421, pp. 33–47. Springer, 2007.
27. A. Margheri, R. De Nicola, and F. Tiezzi. Orchestrating Tuple-based Languages. Technical report, Univ. Firenze, 2011. http://rap.dsi.unifi.it/korc/korcTR.pdf.
28. 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.
29. 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.
30. 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.
31. OASIS WSBPEL TC. Web Services Business Process Execution Language Version 2.0. Technical report, OASIS, April 2007.
32. C. Peltz. Web Services Orchestration and Choreography. *Computer*, 36(10):46–52, 2003.
33. 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.
34. M. Wand and I. Siveroni. Constraint systems for useless variable elimination. In *POPL*, pp. 291–302. ACM, 1999.
35. I. Wehrman, D. Kitchin, W.R. Cook, and J. Misra. A timed semantics of orc. *Theor. Comput. Sci.*, 402(2-3):234–248, 2008.